UNIVERSITÉ PARIS SUD

Comprendre le monde, construire l'avenir®

UNIVERSITÉ PARIS-SUD

ÉCOLE DOCTORALE D'INFORMATIQUE

Laboratoire de Recherche en Informatique

THÈSE DE DOCTORAT

Renforcement du Noyau d'un Démonstrateur SMT

Conception et Implantation de Procédures de Décisions Efficaces

---

soutenue le 10 juin 2013

par

Mohamed IGUERNELALA

Commission d'examen:

Frédéric BESSON	-	Examinateur
Alessandro CIMATTI	-	Rapporteur
Sylvain CONCHON	-	Encadrant, directeur de thèse
Évelyne CONTEJEAN	-	Co-encadrante
Florent HIVERT	-	Président du jury
Michaël RUSINOWITCH	-	Rapporteur
Ralf TREINEN	-	Examinateur

PHD THESIS

Strengthening the Heart of an SMT-Solver

Design and Implementation of Efficient Decision Procedures

Mohamed IGUERNELALA

[version 3 — compiled on July 6, 2013]

- Acknowledgments -

I wish to express my sincere thanks to my advisors, Sylvain Conchon and Évelyne Contejean. Their help was very valuable for the accomplishment of this work.

I would like to thank current and former members of TOCCATA (previously PROVAL) team. It was very helpful for me to work in a pleasant ambiance surrounded by friendly people.

Enormous thanks to my parents, my sisters and my wife. Their encouragements and support helped me stay the course.

Contents

CHAPTER 1

Introduction

Computer systems are now used in various areas, such as medicine, transportation and nuclear power plants. An error in these systems may have consequences that range from occasional discomfort to financial or human loss. For instance, a design bug [103] in PENTIUM 4 cost INTEL 500 million dollars. The explosion of ARIANE 5 due to a bug [87] in its navigation system, cost 370 million dollars. Moreover, several patients died [86] after having received an abnormal dose of radiations, because of a software bug in a radiotherapy machine.

In order to improve hardware and software reliability, several formal methods have been designed. These approaches include testing [119], model checking [9, 52] and deductive verification [23, 63, 12]. Testing techniques are popular in industry, because they are automatic and scale well in practice. Model checkers are now used in hardware design [9, 100, 68] and deductive verification is used in software development [115, 11].

The techniques mentioned above generate a large number of logical formulas. In testing, these formulas are used to characterize feasible paths in control flow graphs. In model checking, they encode states of systems and transition relations. In deductive verification, they are generated from programs annotated with their formal specifications.

In general, formulas issued from software verification contain dozens of ground hypothesis and hundreds of quantified axioms. But, only few instances of some axioms are needed to prove their validity. Furthermore, these formulas mix — in a non-trivial way — boolean connectives with some specific theories, such as linear arithmetic, the free theory of equality and the theory of arrays.

Interactive theorem provers [118, 71] can naturally be used to prove these formulas, especially when the proofs require inductive reasoning. Nonetheless, these tools are very tedious, as the user has to provide a lot of details and plenty of time, which makes them not scalable in practice. Thus, automatic theorem provers (ATPs) seem to be a good alternative, since they allow a high degree of automation and scalability on large projects.

SAT solvers [58, 92, 112] designate a family of automatic tools that targets reasoning in propositional logic. These tools have made a spectacular progress and reached a high level of maturity during the two last decades. However, they are not really suitable for proving formulas coming from software verification, because they do not handle quantifiers and theories reasoning.

Conversely, TPTP[1] provers [122, 91, 106] are a generic family of automatic tools that handles reasoning in first-order logic. These ATPs are commonly based on first-order resolution, tableaux methods or superposition calculus. Moreover, they are refutationally complete and provide built-in support for some equational theories, such as the free theory of equality and the AC[2] theory. Internally, they use term rewriting, unification and completion as computation and inference techniques. However, these provers have not succeeded in the context of software verification. The main reason is the lack of efficient reasoning in a combination of interesting theories, such as linear arithmetic.

Halfway between these two families, another technology, called *Satisfiability Modulo Theories* (SMT), has emerged in the last decade. Logical formulas issued from software verification are within the reach of these ATPs. In this thesis, we will focus on this family of automatic provers.

1.1 Satisfiability Modulo Theories

SMT solvers handle formulas expressed in syntactic fragments of first-order logic where some symbols are assigned additional meanings in some specific theories. For instance, the formula

$$(a \leq b + 0 \ \wedge \ a \geq b) \ \Rightarrow \ f(a) = f(b)$$

is a mixture of boolean connectives ($\wedge$, $\Rightarrow$), uninterpreted symbols (a, b, f), an equality predicate ($=$) and arithmetic symbols ($\leq$, $\geq$, $+$, 0). This formula is valid in the combination of the free theory of equality and linear integer arithmetic.

Internally, SMT solvers combine specialized powerful "little engines of proofs" to enable efficient propositional reasoning modulo interesting specific theories. Furthermore, some of them support quantified formulas. The architecture of a basic SMT solver is shown in Figure 1.1. The preprocessor includes operations, such as parsing, type-checking, simplification, learning and CNF conversion. In the rest of this section, we will describe the other components in more detail.

[1] Thousands of Problems for Theorem Provers [117]

[2] The theory of associative and commutative function symbols.

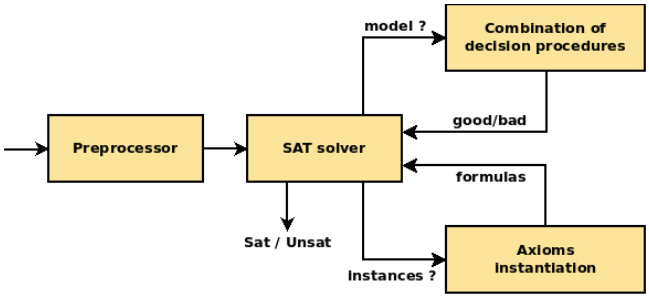

Figure 1.1: The basic architecture of an SMT solver

1.1.1 SAT solving

SAT solvers handle reasoning in propositional logic. They are used to determine whether the propositional variables appearing in a given formula admit a boolean assignment making this formula satisfiable. The main approaches used for that purpose are BDDs [3], the resolution method [108], the DPLL algorithm [40] and the CDCL procedure [112]. The two last techniques are the most popular in SMT. They work on formulas in Conjunctive Normal Form (CNF).

DPLL is a backtracking search procedure that attempts to construct a boolean model for a given formula by iteratively (1) fixing an unassigned variable to a truth value and by (2) deducing all of its consequences. When a conflicting configuration is encountered, backtracking is performed and the last assignment is flipped.

The CDCL algorithm improves DPLL in many ways. First, it uses new data structures for clauses representation that dramatically speed up the deduction of consequences. Second, the choice of the next variable to be assigned is guided by a dynamic variables activity (VSIDS) heuristic [92]. Most importantly, an implied clause is learned after each conflict and the backtracking process is entirely guided by this clause. A very naive implementation of CDCL is sketched in Figure 1.2.

```
1  procedure naive_cdcl() =
2    while true do
3      propagate();
4      if conflict() then
5        begin
6          if no_decisions() then return UNSAT;
7          analyze_conflict();
8          learn_a_clause();
9          backjump(); // non-chronological backtrack
10       end
11     else
12       begin
13         if full_model() then return SAT;
14         decide();
15       end
16   done
```

Figure 1.2: The simplified CDCL procedure

The extension of an abstract DPLL/CDCL procedure with theory reasoning is described in [98]. In earlier approaches, the decisions procedures were only asked to check the satisfiability of full boolean models constructed by the SAT. Modern integration schemes are much more tight. Figure 1.3 sketches such an integration scheme: each time the boolean candidate model is augmented with some assignments, the procedure checks its consistency modulo theory (line 4). Moreover, an incomplete, but fast learning mechanism is used to deduce consequences that are implied at the theory level by the current partial propositional model.

```
a 1  procedure propagate() =
  2    while propagation_stack_not_empty() do
  3      boolean_propagation();
  4      theory_assume();
  5      theory_learn();
  6    done
```

Figure 1.3: Integration of theory reasoning in a DPLL/CDCL

1.1.2 Decision procedures and their combination

Decision procedures are designed to efficiently reason modulo specific theories. These dedicated proofs engines are predictable compared to generic deduction mechanisms. For instance, the congruence closure algorithm [97, 6] and the ground completion procedure [114] are used to decide quantifier-free formulas modulo the free theory of equality. The simplex algorithm [39] and the Fourier-Motzkin method [76] are used to decide quantifier-free formulas modulo linear rational arithmetic. The Omega-Test procedure [104] and various simplex extensions [110] are used to solve linear integer arithmetic.

In many domains, formulas are expressed through a combination of several theories. Consequently, reasoning in the union of these theories is mandatory to decide such formulas. Given two theories $\mathcal{T}_1$ and $\mathcal{T}_2$ equipped with two decision procedures $\mathcal{D}_1$ and $\mathcal{D}_2$ respectively, the combination problem of $\mathcal{T}_1$ and $\mathcal{T}_2$ consists in building — if possible — a decision procedure $\mathcal{D}$ for the union $\mathcal{T}_1 \cup \mathcal{T}_2$ using the individual procedures $\mathcal{D}_1$ and $\mathcal{D}_2$.

Designing efficient frameworks to combine decision procedures is an active research domain. The two historical algorithms for combining theories without shared symbols are Nelson-Oppen's framework [94] and Shostak's method [111].

Nelson-Oppen's framework is very generic. It combines decision procedures for signature-disjoint theories that are stably infinite. For convex theories, it (1)

purifies the given mixed formula using variables abstraction; (2) solves each pure formula using the corresponding decision procedure; (3) exchanges entailed equalities between shared variables. The last point is crucial for completeness, as the individual procedures have to agree on equalities between shared variables to merge their models.

The non-deterministic version handles non-convex theories. It works by guessing implied equalities between shared variables instead of asking the procedures to deduce them all. However, its complexity is much higher. Let us illustrate this method on the following mixed formula:

$$0 \leq a \ \wedge \ a \leq 1 \ \wedge \ f(a) \neq f(0) \ \wedge \ g(a) \neq g(1)$$

The purification step yields the equivalent conjunction:

$$\begin{array}{lll} 0 \leq x \ \wedge \ x \leq 1 \ \wedge \ y = 0 \ \wedge \ z = 1 & & (\varphi_1) \\ \wedge & & \\ x = a \ \wedge \ f(x) \neq f(y) \ \wedge \ g(x) \neq g(z) & & (\varphi_2) \end{array}$$

where x, y and z are abstraction variables, φ_1 is pure in linear integer arithmetic and φ_2 is pure in the free theory of equality. In the second step, we realize that $(0 \leq x \wedge x \leq 1)$ is equivalent to $(x = 0 \ \vee \ x = 1)$ modulo linear integer arithmetic. But, if $x = 0$ (*resp.* $x = 1$), the equality $x = y$ (*resp.* $x = z$) has to be propagated to the decision procedure of the free theory of equality. This implies that $f(x) = f(y)$ (*resp.* $g(x) = g(z)$), which causes an inconsistency.

A substantial amount of work aims at improving this framework. For instance, *Delayed Theories Combination* [27] delegates the combination task to the SAT solver. *Model Based Theories Combination* [46] attempts to reduce the number of propagated equalities while preserving completeness. These methods, are used in several SMT solvers, such as CVC3 [19], CVC4 [14], MathSAT5 [66] and Z3 [43].

Shostak's combination approach is a specific framework for reasoning in the union of the free theory of equality with a signature-disjoint *Shostak theory*. That is, an equational theory that admits a *canonizer* algorithm for computing normal forms of terms modulo theory, and a *solver* routine that transforms equalities into substitutions. For example, the equational parts of linear arithmetic, the theory of records, and the theory of fixed-sized bit-vectors meet these criteria.

This approach handles formulas of the form $(s_1 = t_1 \wedge \cdots \wedge s_n = t_n) \Rightarrow s = t$. It interleaves canonization, solving equalities and rewriting, in order to transform the conjunction $(s_1 = t_1 \wedge \cdots \wedge s_n = t_n)$ into a convergent rewriting system modulo

the Shostak theory. Then, it rewrites and canonizes s and t, to check whether $s = t$ follows. Let us illustrate this mechanism on the following example:

$$(x = 2y \ \wedge \ f(y) - \frac{x}{2} = 0) \ \Rightarrow \ f(x - f(y)) = y$$

Solving the first equality simply returns $x \mapsto 2y$. The second equality is then rewritten to $f(y) - \frac{2y}{2} = 0$ and canonized to $f(y) - y = 0$. Solving the last equality returns $f(y) \mapsto y$. Finally, rewriting and canonizing $f(x - f(y))$ yields y.

The initial formulation of Shostak's framework [111] was neither complete nor terminating. After several attempts, it was completely corrected in [109]. Moreover, Shostak claimed that it would be possible to combine individual canonizers and solvers. While the first assertion was proved correct, it has been shown that solvers do not combine in general [35]. This framework and its variants are used in several tools, such as PVS [101], SVC [16], ICS [60] and our ALT-ERGO SMT solver [21].

1.1.3 Handling quantifiers

Stricto sensu, SMT refers to the extension of a SAT solver with built-in decision procedures to reason modulo theories. However, in many application domains, formulas contain both ground and quantified parts. For instance, in deductive software verification, quantified formulas are used to encode memory models of programming languages and some properties assessed in programs specification.

In general, SMT solvers with quantifiers support use instantiation techniques. These techniques are based on matching modulo ground equalities and the notion of triggers. Furthermore, some SMT solvers integrate superposition calculus [44] to overcome the limitations of matching techniques.

1.2 The SMT revolution

Satisfiability Modulo Theories is a young research topic. First developments date back to Nelson's PhD thesis [95] in late 1970s — early 1980s. During the last years, we have witnessed an important improvement of this technology up to the present time, at which SMT solvers are very popular. They are used in various domains, such as hardware design, software verification, model-checking, symbolic execution, test-case generation, etc.

However, there are still a lot of challenges in SMT solving. For instance, these include the design of new procedures for the theory of floating-point numbers and

set theory, the improvement of existing procedures, the enhancement of theories combination schemes and a better handling of quantifiers.

There is an active community around SMT. In 2003, the SMT-LIB [105] initiative was created to provide standard descriptions of some interesting theories, to define a common language format, and to collect benchmarks. Moreover, SMT-competition [15], SMT-workshop and the SAT/SMT summer school are yearly events devoted to this topic. Figure 1.4 summarizes some advances in SAT solving, SMT solving, and decision procedures design and combination.

Period	Advances and tools	Remarks
2010	The CVC4 SMT solver [14]	Successor of CVC3
2010	The MathSAT5 SMT solver [29]	Successor of MathSAT4
2007	The Z3 SMT solver [43]	
2007	The MathSAT4 SMT solver [28]	Successor of MathSAT3
2006	The CVC3 SMT solver [19]	Successor of CVC
2006	The ALT-ERGO SMT solver [21]	Shostak-based
2005	The Yices SMT solver [56]	
2005	The MathSAT3 SMT solver [25]	
2005	SMT-COMP [15]	The first edition of the competition
2005	The Simplify SMT solver [51]	Public release
2004	The CVC-Lite SMT solver [13]	Successor of CVC
2003	The MiniSat SAT solver [58]	Small and readable but powerful
2003	The SMT-LIB initiative	Common language and benchmarks
2002	The Math-SAT solver [4]	
2002	The CVC SMT solver [20]	Successor of SVC, based on Chaff
2002	ESC/Java	A static programs checker that uses Simplify
2001	The ICS solver [60]	Shostak-based
2001	The Chaff SAT solver [92]	The VSIDS decision heuristic
1998	ESC/Modula-3	A static programs checker that uses Simplify
1996	The SVC SMT solver [16]	Shostak-based
1996	The GRASP SAT solver [112]	Non chronological backtracking and clause learning
1992	The PVS verification system [101]	Uses procedures combined with the Shostak method
1984	Shostak's combination framework [111]	Procedures combination framework
1980	Nelson-Oppen's framework [96]	Procedures combination framework
1979	The Standford Pascal Verifier [88]	Program verification system that uses the Simplifier
1979	The Simplifier decision procedure [93]	Based on the Nelson-Oppen's framework
1962	The DPLL framework [40]	SAT solving

Figure 1.4: A summary of some advances in the domains of decision procedures design and combination, SAT solving and SMT solving.

1.3 Overview of the contributions

This thesis addresses the enhancement of our Alt-Ergo theorem prover to make it effective and usable in the context of software verification. We summarize in this section our main contributions.

1.3.1 Quantifier-free linear integer arithmetic

The theory of quantifier-free linear integer arithmetic (QF-LIA) is ubiquitous in many domains, such as verification, linear programming, compiler optimization, planning and scheduling. In particular, it constitutes — together with the free theory of equality — a must-have in deductive software verification. Most of the procedures implemented in state-of-the-art SMT solvers are extensions of either the simplex algorithm [42, 14, 66, 45] or the Fourier-Motzkin method [19, 21]. Both techniques first relax the initial problem to the rational domain and then proceed by branching / cutting methods or by projection.

Simplex extensions, such as *branch-and-bound* and *cutting-planes* [110], are often drowned in a huge search space, when they come to perform a case-split analysis. Furthermore, they are not complete *w.r.t.* the deduction of all entailed equalities. Conversely, Fourier-Motzkin extensions, such as *Omega-Test* [104], do not scale in practice. In fact, they potentially introduce a double exponential number of intermediate inequalities, which saturates the memory.

The first contribution of this thesis is a novel procedure for deciding QF-LIA. Roughly speaking, given a conjunction of inequalities $\bigwedge_i \sum_j a_{i,j}\, x_j + b_i \leq 0$ over integers, our algorithm attempts to infer precise lower bounds for the affine forms $a_{i,j}\, x_j + b_i$, in order to cut down the search space. These bounds are computed using a rational simplex to solve auxiliary linear optimization problems. These problems simulate particular runs of the Fourier-Motzkin algorithm.

A key feature of our method is that we can safely deduce the satisfiability of the original problem, if none of the affine forms have a lower bound. This is clearly a positive point compared to traditional simplex-based extensions. Moreover, we can infer all implied (disjunctions of) equalities for the affine forms $a_{i,j}\, x_j + b_i$ that admit a lower bound. In addition, the use of a rational simplex in practice allows us to circumvent the scalability issue of Fourier-Motzkin extensions.

Since our technique is based on bounds inference and maintenance, it is easily extensible with intervals calculus. In practice, this allows us to incorporate non-linear arithmetic reasoning. Note that equality constraints are eliminated using solving and substitution techniques *à la* Omega-Test.

1.3.2 Preprocessing and SAT solving

To validate our procedure for linear integer arithmetic, we have conducted some experiments with ALT-ERGO, using the QF-LIA benchmark [18]. Unfortunately, we were quickly faced with the complexity of certain formulas in this test-suite. In fact, these formulas have a rich propositional structure and heavily use some high level constructs, such as LET-IN and IF-THEN-ELSE. Although sufficient in the context of deductive software verification, the preprocessor and the SAT solver of ALT-ERGO are not suitable for this kind of formulas.

To circumvent this issue, we implemented a small SMT solver dedicated to quantifier-free linear integer arithmetic. This prototype, called CTRL-ERGO, is built upon an OCAML reimplementation of MINISAT [57]. Its preprocessor incorporates a new contextual simplification step to minimize formulas containing LET-IN and IF-THEN-ELSE constructs. In addition, the QF-LIA procedure implements some features, such as theory propagation and dynamic clustering.

1.3.3 Built-in AC reasoning modulo theories

Many mathematical operators occurring in automated reasoning enjoy the associativity and commutativity (AC) properties. Examples of such operators include union and intersection of sets, boolean and bit-vectors operators (and, or, xor), and arithmetic operators (addition, linear and non-linear multiplication).

The easiest way to handle these properties in an SMT solver is to add the axioms below in its instantiation engine for each function symbol u that is AC.

$$\forall x.\forall y.\forall z. \quad u(x,u(y,z)) \;=\; u(u(x,y),z) \qquad \text{(A)}$$

$$\forall x.\forall y. \quad\quad u(x,y) \;=\; u(y,x) \qquad\quad \text{(C)}$$

Unfortunately, this method does not work in practice. Indeed, the mere addition of these axioms to a prover will usually glut it with plenty of intermediate terms and equalities which will strongly impact its performances.

While SMT solvers provide built-in support for some specific AC symbols, such as linear arithmetic and boolean operators, they rather rely on axiomatization to deal with generic user-defined AC symbols. Conversely, AC reasoning is well investigated in the rewriting community. For instance, the AC completion procedure, implemented at the heart of some TPTP provers, enables a powerful generic treatment of AC symbols. Furthermore, when the input problem has no variables, AC completion provides a decision algorithm [89] for the combination of the free theory of equality and the AC theory.

In many application domains, AC is only a part of the automated deduction problem. What we really need is to decide formulas combining AC symbols and other theories. For instance, a combination of AC reasoning with linear arithmetic and the free theory equality is necessary to prove the formula

$$\left(\begin{array}{l} u(a, c_2 - c_1) = a \ \wedge \ d = c_1 + 1 \ \wedge \\ u(e_1, e_2) - f(b) = u(d, d) \ \wedge \ e_2 = b \ \wedge \\ u(b, e_1) = f(e_2) \ \wedge \ c_2 = 2\, c_1 + 1 \end{array}\right) \ \Rightarrow \ a = u(a, 0)$$

where u is an AC symbol, the symbols $+, -, 0, 1, 2$ belong to the theory of linear arithmetic, and the rest of the symbols are uninterpreted.

In this thesis, we have investigated the incorporation of generic built-in AC reasoning in an SMT solver. For this purpose, we have designed a framework, called AC(X), to reason in the combination of the free theory of equality, the AC theory and an arbitrary signature-disjoint Shostak theory.

The AC theory cannot be directly combined using Shostak's framework, since it does not provide a solver routine. To circumvent this limitation, we have adopted a slightly different approach: we followed Shostak's combination technique and extended the ground AC completion procedure with an arbitrary signature-disjoint Shostak theory. This modular and non-intrusive extension rests on the integration of the canonizer and the solver routines, provided by the Shostak theory in the ground AC completion procedure.

Note that, at about the same period as our investigations, Tiwari has studied [120] the combination of AC reasoning with the free theory of equality and the theory of polynomial rings in the Nelson-Oppen framework. However, no implementation or experimentation were given. In addition, our philosophy in ALT-ERGO is to rather favor Shostak-like methods to combine convex equational theories.

1.3.4 Implementation in ALT-ERGO

ALT-ERGO [21] is an SMT solver used to discharge logical formulas issued from deductive programs verification. It is currently used in tools, such as Why3 [61], Caveat [115], Frama-C [63], Spark [11] and GNATprove [67].

During this thesis, we enhanced the core of ALT-ERGO in many ways. First, we extended our solver with the AC(X) framework. Currently, AC(X) is used to handle the associativity and the commutativity properties of both user-defined AC symbols and non-linear multiplication. Moreover, a subtle interaction between

linear arithmetic and AC allows us to partially support the distributivity property of non-linear multiplication over addition. Second, we have implemented a state-of-the-art decision procedure [64] for the theory of functional arrays with extentionality and a procedure for the theory of enumerated data types.

1.4 Outline

This dissertation is organized follows: Chapter 2 introduces some preliminaries and background material that will be used in the rest of this thesis. In Chapter 3, we describe our procedure for deciding the theory of quantifier-free linear integer arithmetic and provide some implementation details about the CTRL-ERGO SMT solver. In Chapter 4, we present our AC(X) framework that enables reasoning in the combination of the free theory of equality, the AC theory and a signature-disjoint Shostak theory. We also describe some prospective extensions of AC(X) and discuss the issues we encountered. Chapter 5 describes the core of ALT-ERGO and the various extensions we have made to enhance it. Finally, in Chapter 6, we review some interesting improvements and extensions of our work and conclude.

CHAPTER 2

Background

In this chapter, we first recall some notations and definitions about many-sorted first-order logic, satisfiability modulo theories and Shostak theories. Then, we present some interesting first-order theories in SMT.

2.1 Many-sorted first-order logic

2.1.1 Syntax

Signature

In many-sorted first-order logic, a signature Σ is a tuple $(\mathcal{F}, \mathcal{P}, \mathcal{S}, \tau_{\mathcal{F}}, \tau_{\mathcal{P}})$, where:

- $\mathcal{F}$ and $\mathcal{P}$ are two disjoint sets of function and predicate symbols, respectively
- $\mathcal{S}$ is a set of sort symbols
- $\tau_{\mathcal{F}} : \mathcal{F} \to \mathcal{S}^+$ associates to each symbol $f \in \mathcal{F}$ a tuple $s_1 \times \cdots \times s_n \times s$, usually denoted $s_1 \times \cdots \times s_n \to s$
- $\tau_{\mathcal{P}} : \mathcal{P} \to \mathcal{S}^*$ associates to each symbol $p \in \mathcal{P}$ a tuple $s_1 \times \cdots \times s_n$

The tuple $s_1 \times \cdots \times s_n \to s$ (*resp.* $s_1 \times \cdots \times s_n$) is called the sort (arity) of the symbol f (*resp.* p). A constant is a function (*resp.* predicate) symbol with arity $s \in \mathcal{S}$.

In addition to the symbols in $\mathcal{P}$, we dispose of a binary predicate $\approx_s$ of sort $s \times s$, for each $s \in \mathcal{S}$. These predicates, that represent syntactic equality, will be denoted $\approx$ when the sort information is irrelevant or deductible from the context.

Terms

Let Σ be a signature, $\mathcal{X}$ a set of variables disjoint from $\mathcal{F}$ and $\mathcal{P}$, and $\tau_{\mathcal{X}} : \mathcal{X} \to \mathcal{S}$ a mapping from variables to sorts. The set $\mathcal{T}_\Sigma(\mathcal{X})$ of well-sorted terms is given by:

$$\mathcal{T}_\Sigma(\mathcal{X}) = \bigcup_{s \in \mathcal{S}} \mathcal{T}^s_\Sigma(\mathcal{X})$$

where $\mathcal{T}^s_\Sigma(\mathcal{X})$ is the set of well-sorted terms of sort s defined as follows:

- $x \in \mathcal{T}^s_\Sigma(\mathcal{X})$ if $x \in \mathcal{X}$ and $\tau_{\mathcal{X}}(x) = s$
- $f(t_1, \cdots, t_n) \in \mathcal{T}^s_\Sigma(\mathcal{X})$ if $f \in \mathcal{F}$ and $\tau_{\mathcal{F}}(f) = s_1 \times \cdots \times s_n \to s$ and $t_i \in \mathcal{T}^{s_i}_\Sigma(\mathcal{X})$, for each $1 \leq i \leq n$

Atoms

We then define the set $\mathcal{AT}_\Sigma(\mathcal{X})$ of well-sorted atoms as follows:

- $t_1 \approx_s t_2 \in \mathcal{AT}_\Sigma(\mathcal{X})$ if $s \in \mathcal{S}$ and $t_1, t_2 \in \mathcal{T}^s_\Sigma(\mathcal{X})$
- $p(t_1, \cdots, t_n) \in \mathcal{AT}_\Sigma(\mathcal{X})$ if $p \in \mathcal{P}$ and $\tau_{\mathcal{P}}(p) = s_1 \times \cdots \times s_n$ and $t_i \in \mathcal{T}^{s_i}_\Sigma(\mathcal{X})$, for each $1 \leq i \leq n$

Formulas

On top of atoms, we define the set $\mathcal{FO}_\Sigma(\mathcal{X})$ of well-sorted formulas as follows:

- $\mathcal{AT}_\Sigma(\mathcal{X}) \subseteq \mathcal{FO}_\Sigma(\mathcal{X})$
- $(\neg\, \varphi) \in \mathcal{FO}_\Sigma(\mathcal{X})$ if $\varphi \in \mathcal{FO}_\Sigma(\mathcal{X})$
- $(\varphi_1 \bowtie \varphi_2) \in \mathcal{FO}_\Sigma(\mathcal{X})$ if $\varphi_1 \in \mathcal{FO}_\Sigma(\mathcal{X})$ and $\varphi_2 \in \mathcal{FO}_\Sigma(\mathcal{X})$ and $\bowtie \in \{ \wedge, \vee, \Rightarrow, \Leftrightarrow \}$
- $(Q\, x : s.\ \varphi) \in \mathcal{FO}_\Sigma(\mathcal{X})$ if $x \in \mathcal{X}$ and $\tau_{\mathcal{X}}(x) = s$ and $\varphi \in \mathcal{FO}_\Sigma(\mathcal{X})$ and $Q \in \{ \forall, \exists \}$

In the rest of this thesis, we will use (possibly with subscripts) the symbols a, b, f, g, u to denote function symbols, p, q to denote predicate symbols, s, t, l, r to denote terms, x, y, z to denote variables, and $\varphi,\ \psi$ to denote formulas. Depending on the context, we will also use a to denote atoms, l to denote literals and s to denote sorts.

Sort annotations used for quantified formulas (i.e. $Q\ x : s.\ \varphi$) will be omitted when they are irrelevant or deductible from the context. For instance, we simply write $\forall x.\ x \approx_s x$ instead of $\forall x : s.\ x \approx_s x$, where $s \in \mathcal{S}$. We will also use the abbreviation $\forall x_1, \cdots, x_n.\ \varphi$ to denote the formula $\forall x_1.\ \cdots \forall x_n.\ \varphi$

Viewing terms as trees, subterms within a term s are identified by their positions. $\mathcal{P}os(t)$ denotes the set of all positions of a term t. Given a position p, $s|_p$ denotes the subterm of s at position p, and $s[r]_p$ the term obtained by replacement of $s|_p$ by the term r. We will also use the notation $s(p)$ to denote the symbol at position p in the tree. The root position is denoted by Λ. Given a subset $\mathcal{F}'$ of $\mathcal{F}$, a subterm $t|_p$ of t is a $\mathcal{F}'$-alien of t if $t(p) \notin \mathcal{F}'$ and p is minimal *w.r.t* the prefix word ordering. Notice that, according to this definition, a variable is a $\mathcal{F}'$-alien. The multiset of $\mathcal{F}'$-aliens of a term t will be denoted $\mathcal{A}_{\mathcal{F}'}(t)$.

A literal is a formula of the form a or $\neg a$, where a is an atom. A clause is a disjunction $\bigvee_i l_i$ of literals. The variables appearing in a clause — if any — are implicitly universally quantified. A formula in conjunctive normal form (CNF) is a conjunction $\bigwedge_j(\bigvee_i l_{ij})$ of clauses.

Variables

The set $vars(t)$ of variables occurring in a term t is defined as follows:

- $vars(x) = \{x\}$ if $x \in \mathcal{X}$
- $vars(f(t_1, \cdots, t_n)) = \bigcup_{i=1}^{n} vars(t_i)$ if $f \in \mathcal{F}$ and $\tau_{\mathcal{F}}(f) = s_1 \times \cdots \times s_n \to s$

A term t is *ground* if $vars(t) = \emptyset$. The set of all ground terms is denoted $\mathcal{T}_\Sigma$. The set $vars_{\mathcal{A}}(a)$ of variables occurring in an atom a is defined in a similar way. The set $\mathcal{F}vars(\varphi)$ of *free* variables occurring in a formula φ is defined by:

- $\mathcal{F}vars(\varphi') = vars_{\mathcal{A}}(\varphi')$ if $\varphi' \in \mathcal{AT}_\Sigma(\mathcal{X})$
- $\mathcal{F}vars(\neg\, \varphi') = \mathcal{F}vars(\varphi')$ if $\varphi' \in \mathcal{FO}_\Sigma(\mathcal{X})$
- $\mathcal{F}vars(\varphi_1 \bowtie \varphi_2) = \mathcal{F}vars(\varphi_1) \cup \mathcal{F}vars(\varphi_2)$ if $\varphi_1 \in \mathcal{FO}_\Sigma(\mathcal{X})$ and $\varphi_2 \in \mathcal{FO}_\Sigma(\mathcal{X})$ and $\bowtie \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$
- $\mathcal{F}vars(Q\ x.\ \varphi') = \mathcal{F}vars(\varphi') \setminus \{x\}$ if $x \in \mathcal{X}$ and $\varphi' \in \mathcal{FO}_\Sigma(\mathcal{X})$ and $Q \in \{\, \forall,\ \exists \,\}$

The set $\mathcal{B}vars(\varphi)$ of *bounded* variables occurring in φ is defined by:

- $\mathcal{B}vars(\varphi) = \emptyset$ if $\varphi \in \mathcal{AT}_\Sigma(\mathcal{X})$
- $\mathcal{B}vars(\neg\, \varphi) = \mathcal{B}vars(\varphi)$ if $\varphi \in \mathcal{FO}_\Sigma(\mathcal{X})$
- $\mathcal{B}vars(\varphi_1 \bowtie \varphi_2) = \mathcal{B}vars(\varphi_1) \cup \mathcal{B}vars(\varphi_2)$ if $\varphi_1 \in \mathcal{FO}_\Sigma(\mathcal{X})$ and $\varphi_2 \in \mathcal{FO}_\Sigma(\mathcal{X})$ and $\bowtie \ \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$
- $\mathcal{B}vars(Q\ x.\ \varphi) = \mathcal{B}vars(\varphi) \cup \{x\}$ if $x \in \mathcal{X}$ and $\varphi \in \mathcal{FO}_\Sigma(\mathcal{X})$ and $Q \in \{\ \forall,\ \exists\ \}$

The formula φ is a *sentence* (or is *ground*) if $\mathcal{F}vars(\varphi) = \emptyset$. It is *quantifier-free* if $\mathcal{B}vars(\varphi) = \emptyset$. Note that, quantifier-free sentences contain only ground terms. The definitions of $\mathcal{V}ars$, $\mathcal{V}ars_{\mathcal{A}}$, $\mathcal{F}vars$ and $\mathcal{B}vars$ given above are generalized to sets of terms, atoms and formulas in the usual way.

Substitutions

Let Σ be a signature and $\mathcal{X}$ a set of variables disjoint from $\mathcal{F}$ and $\mathcal{P}$. A substitution $\sigma : \mathcal{X} \to \mathcal{T}_\Sigma(\mathcal{X})$ is a sort-preserving mapping such that $\sigma(x) \neq x$ for finitely many variables $x \in \mathcal{X}$.
Let σ be a substitution. The domain of σ is the set $\mathcal{D}om(\sigma)$ of variables defined by:

$$\mathcal{D}om(\sigma) = \{\ x \in \mathcal{X} \ \mid\ \sigma(x) \neq x\ \}$$

The range of σ is the set $\mathcal{R}an(\sigma)$ of terms defined by:

$$\mathcal{R}an(\sigma) = \{\ \sigma(x) \ \mid\ x \in \mathcal{D}om(\sigma)\ \}$$

The variables range of σ is the set $\mathcal{VR}an(\sigma)$ of variables occurring in the terms of σ's range. It is given by:

$$\mathcal{VR}an(\sigma) = \bigcup_{t \in \mathcal{R}an(\sigma)} \mathcal{V}ars(t)$$

Given a substitution σ such that $\mathcal{D}om(\sigma) = \{x_1, \cdots, x_n\}$, we usually write σ as a partial mapping of the form $\{\ x_1 \mapsto \sigma(x_1),\ \cdots,\ x_n \mapsto \sigma(x_n)\ \}$. A *variable renaming* over a subset $\mathcal{X}'$ of $\mathcal{X}$ is a substitution from variables to variables such that $x = y$ whenever $\sigma(x) = \sigma(y)$, for each $x, y \in \mathcal{X}'$.

The application of a substitution σ on a term t is denoted $t\sigma$. It consists in simultaneously replacing each variable $x \in \mathcal{V}ars(t)$ with $\sigma(x)$. More formally:

- $x\sigma = \sigma(x)$ if $x \in \mathcal{X}$
- $f(t_1, \cdots, t_n)\sigma = f(t_1\sigma, \cdots, t_n\sigma)$ if $f \in \mathcal{F}$ and $\tau_{\mathcal{F}}(f) = s_1 \times \cdots \times s_n \to s$

Applying σ on a formula φ is also denoted $\varphi\sigma$. It consists in simultaneously replacing the occurrences of each variable $x \in \mathcal{F}vars(\varphi)$ with $\sigma(x)$. However, one should pay attention to capture issues, as the intersection $\mathcal{F}vars(\varphi) \cap \mathcal{B}vars(\varphi)$ is not necessarily empty. In fact, every occurrence of the variables introduced by the application of σ should be free in $\varphi\sigma$. More formally:

- $(t_1 \approx_s t_2)\sigma = (t_1\sigma) \approx_s (t_2\sigma)$ if $s \in \mathcal{S}$ and $t_1, t_2 \in \mathcal{T}^s_\Sigma(\mathcal{X})$
- $p(t_1, \cdots, t_n)\sigma = p(t_1\sigma, \cdots, t_n\sigma)$ if $p \in \mathcal{P}$ and $\tau_{\mathcal{P}}(p) = s_1 \times \cdots \times s_n$
- $(\neg\, \varphi)\sigma = \neg\, (\varphi\sigma)$ if $\varphi \in \mathcal{FO}_\Sigma(\mathcal{X})$
- $(\varphi_1 \bowtie \varphi_2)\sigma = (\varphi_1\sigma) \bowtie (\varphi_2\sigma)$ if $\varphi_1, \varphi_2 \in \mathcal{FO}_\Sigma(\mathcal{X})$ and $\bowtie \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$
- $(Q\ x.\ \varphi)\sigma = Q\ x.\ (\varphi\sigma')$ if $x \in \mathcal{X}$ and $\varphi \in \mathcal{FO}_\Sigma(\mathcal{X})$ and $Q \in \{\ \forall,\ \exists\ \}$ and $\sigma' : \begin{cases} y \mapsto \sigma(y) & \text{for each } y \neq x \\ x \mapsto x \end{cases}$ and $x \notin \mathcal{VR}an(\sigma')$

If $x \in \mathcal{VR}an(\sigma')$, a variable renaming $\{x \mapsto z\}$, where $z \notin (\mathcal{F}vars(\varphi) \cup \mathcal{B}vars(\varphi))$, is necessary before applying σ on $Q\ x.\ \varphi$ to ensure the condition $x \notin \mathcal{VR}an(\sigma')$.

The composition of two substitution θ and σ, denoted $\theta \circ \sigma$, is defined by:

$$\begin{array}{rcccl} \theta \circ \sigma & : & \mathcal{X} & \to & \mathcal{T}_\Sigma(\mathcal{X}) \\ & & x & \mapsto & (x\,\sigma)\theta \qquad \text{for each } x \in \mathcal{X} \end{array}$$

A substitution σ is *idempotent* if σ and $\sigma \circ \sigma$ are identical. In this case, we have $\mathcal{D}om(\sigma) \cap \mathcal{VR}an(\sigma) = \emptyset$.

2.1.2 Semantics

In this section, we assume given a signature $\Sigma = (\mathcal{F}, \mathcal{P}, \mathcal{S}, \tau_{\mathcal{F}}, \tau_{\mathcal{P}})$, a set of variables $\mathcal{X}$ disjoint from $\mathcal{F}$ and $\mathcal{P}$ and a mapping $\tau_{\mathcal{X}} : \mathcal{X} \to \mathcal{S}$ from variables to sorts.

Structure

A Σ-structure over $\mathcal{X}$ is a mapping that satisfies the following properties:

- each sort $s \in \mathcal{S}$ is associated with a nonempty domain A_s
- each variable $x \in \mathcal{X}$ of sort s is associated with an element $x^{\mathcal{A}} \in A_s$
- each constant symbol $c \in \mathcal{F}$ of sort s is associated with an element $c^{\mathcal{A}} \in A_s$
- each function symbol $f \in \mathcal{F}$ of sort $s_1 \times \cdots \times s_n \to s$, where $n > 0$, is associated with a function $f^{\mathcal{A}} : A_{s_1} \times \cdots \times A_{s_n} \to A_s$
- each predicate symbol $p \in \mathcal{P}$ of sort $s_1 \times \cdots \times s_n$ is associated with a subset $p^{\mathcal{A}} \subseteq A_{s_1} \times \cdots \times A_{s_n}$

Let $x \in \mathcal{X}$ be a variable. We say that a Σ-structure $\mathcal{A}$ over $\mathcal{X}$ is an x-variant of a Σ-structure $\mathcal{B}$ over $\mathcal{X}$ if the restrictions of $\mathcal{A}$ and $\mathcal{B}$ over $\mathcal{X} \setminus \{x\}$ are identical.

Interpretation

Let $\mathcal{A}$ be a Σ-structure over $\mathcal{X}$ and $t \in \mathcal{T}^s_\Sigma(\mathcal{X})$ a term of sort s. The interpretation of t under $\mathcal{A}$ is denoted $\mathcal{A}(t)$. It is the object $t^{\mathcal{A}} \in A_s$ recursively defined by:

- $\mathcal{A}(x) = x^{\mathcal{A}}$ if $x \in \mathcal{X}$
- $\mathcal{A}(c) = c^{\mathcal{A}}$ if $c \in \mathcal{F}$ and $\tau_{\mathcal{F}}(c) = s$
- $\mathcal{A}(f(t_1, \cdots, t_n)) = f^{\mathcal{A}}(\mathcal{A}(t_1), \cdots, \mathcal{A}(t_n))$ if $f \in \mathcal{F}$ and $\tau_{\mathcal{F}}(f) = s_1 \times \cdots \times s_n \to s$

Let $\varphi \in \mathcal{FO}_\Sigma(\mathcal{X})$ be a formula. The interpretation of φ under $\mathcal{A}$ is denoted $\mathcal{A}(\varphi)$. It is a truth value in $\{true,\ false\}$ recursively defined by:

- $\mathcal{A}(t_1 \approx t_2)$ is $true$ iff $\mathcal{A}(t_1) = \mathcal{A}(t_2)$
  is $false$ otherwise
- $\mathcal{A}(p(t_1, \cdots, t_n))$ is $true$ iff $(\mathcal{A}(t_1), \cdots, \mathcal{A}(t_n)) \in p^{\mathcal{A}}$
  is $false$ otherwise
  where $p \in \mathcal{P}$ and $\tau_{\mathcal{P}}(p) = s_1 \times \cdots \times s_n$
- $\mathcal{A}(\neg\, \varphi)$ is $true$ iff $\mathcal{A}(\varphi)$ is $false$
  is $false$ otherwise
- $\mathcal{A}(\varphi_1 \wedge \varphi_2)$ is $true$ iff $\mathcal{A}(\varphi_1)$ is $true$ and $\mathcal{A}(\varphi_2)$ is $true$
  is $false$ otherwise
- $\mathcal{A}(\varphi_1 \vee \varphi_2)$ is $true$ iff $\mathcal{A}(\varphi_1)$ is $true$ or $\mathcal{A}(\varphi_2)$ is $true$
  is $false$ otherwise
- $\mathcal{A}(\varphi_1 \Rightarrow \varphi_2)$ is $true$ iff $\mathcal{A}(\varphi_1)$ is $false$ or $\mathcal{A}(\varphi_2)$ is $true$
  is $false$ otherwise
- $\mathcal{A}(\varphi_1 \Leftrightarrow \varphi_2)$ is $true$ iff $\mathcal{A}(\varphi_1)$ and $\mathcal{A}(\varphi_2)$ are identical
  is $false$ otherwise
- $\mathcal{A}(\forall x.\, \varphi)$ is $true$ iff $\mathcal{B}(\varphi)$ is $true$ for every x-variant $\mathcal{B}$ of $\mathcal{A}$
  is $false$ otherwise
- $\mathcal{A}(\exists x.\, \varphi)$ is $true$ iff $\mathcal{B}(\varphi)$ is $true$ for some x-variant $\mathcal{B}$ of $\mathcal{A}$
  is $false$ otherwise

Model, satisfiability and validity

Let $\varphi \in \mathcal{FO}_{\Sigma}(\mathcal{X})$ be a formula. If there exists a Σ-structure $\mathcal{A}$ such that φ evaluates to $true$ under $\mathcal{A}$, we say that $\mathcal{A}$ satisfies φ, and we write $\mathcal{A} \models \varphi$. A model for φ is any Σ-structure satisfying it.

A formula φ is *satisfiable* if it has a model. It is *unsatisfiable* if it is not satisfiable. It is *valid*, denoted $\models \varphi$, if every Σ-structure is a model for φ. A formula ψ is a logical consequence of φ if every model of φ is also a model for ψ.

The notions above generalize to sets of formulas as usual, by viewing a set $\Phi \subseteq \mathcal{F}_{\Sigma}(\mathcal{X})$ of formulas as a conjunction $\bigwedge_{\varphi \in \Phi} \varphi$.

2.1.3 First-order theories and satisfiability modulo

A first-order *theory* T over a signature Σ is defined by a set of sentences in $\mathcal{FO}_\Sigma(\mathcal{X})$, called axioms. We usually use the symbol T to denote both the name of the theory and its set of axioms.

A theory T is said *consistent* if its set of axioms is satisfiable, otherwise it is *inconsistent*. In the rest of this thesis, we will use the symbol $\bot$ to indicate that a theory (or a logical formula) is inconsistent.

A formula φ is satisfiable in a theory T, or satisfiable modulo T, or T-satisfiable, if there exists a model $\mathcal{A}$ that satisfies both T and φ. We then write $\mathcal{A} \models_{\mathrm{T}} \varphi$. A formula φ is valid modulo T, or T-valid, if it is a logical consequence of the set of axioms T. In this case, we write $\models_{\mathrm{T}} \varphi$. In order to show that a formula φ is T-valid, SMT solvers rather attempts to prove that $\neg\varphi$ is T-unsatisfiable.

The decision problem modulo a theory takes as input a first-order formula and outputs $true$ if this formula is valid modulo the theory. We say that a first-order theory is *decidable* if there exists an algorithm that solves the decision problem modulo that theory, otherwise it is *undecidable*. The satisfiability problem modulo a theory is defined is a similar way, except that we are interested in satisfiability instead of validity.

Convexity. A theory T over a signature Σ is *convex* if, whenever a disjunction of equalities is a logical consequence of a conjunction of literals, there exists a single equality in this disjunction that is a consequence of the conjunction. More formally, T is convex if the T-validity of any Σ-formula of the form (where l_i are Σ-literals)

$$\models_{\mathrm{T}} (\bigwedge_{i=1}^{n} l_i \Rightarrow \bigvee_{j=1}^{m} u_j \approx v_j)$$

implies the T-validity of the formula below for some integer j such that $0 \leq j \leq m$

$$\models_{\mathrm{T}} (\bigwedge_{i=1}^{n} l_i \ \Rightarrow \ u_j \approx v_j)$$

Stable infinity. A theory T over a signature $\Sigma = (\mathcal{F}, \mathcal{P}, \mathcal{S}, \tau_{\mathcal{F}}, \tau_{\mathcal{P}})$ is *stably infinite* if every T-satisfiable formula admits a model $\mathcal{A}$ in which the cardinality of A_s is infinite, for each sort $s \in \mathcal{S}$.

2.2 Shostak theories

Let $\Sigma = (\mathcal{F}, \mathcal{P}, \mathcal{S}, \tau_{\mathcal{F}}, \tau_{\mathcal{P}})$ be a signature, such that $\mathcal{P} = \emptyset$. Let $\mathcal{X}$ be a set of variables and T a first-order theory over Σ.

A canonizer for T is a function (algorithm)

$$\texttt{canon} \ : \ \mathcal{T}_\Sigma(\mathcal{X}) \ \rightarrow \ \mathcal{T}_\Sigma(\mathcal{X})$$

which returns for every term t_1 a unique representative t_2, called its canonical form, such that $t_1 \approx t_2$ is T-valid. Moreover, the canonizer enjoys the following conditions, for any terms t_1 and t_2:

(1) $\texttt{canon}(\texttt{canon}(t_1)) = \texttt{canon}(t_1)$

(2) $\models_{\mathrm{T}} t_1 \approx t_2$ iff $\texttt{canon}(t_1) = \texttt{canon}(t_2)$

(3) $\mathcal{V}ars(\texttt{canon}(t_1)) \subseteq \mathcal{V}ars(t_1)$

(4) if $\texttt{canon}(t_1) = t_1$ then $\texttt{canon}(t_1|_p) = t_1|_p$ for every position $p \in \mathcal{P}os(t_1)$

These proprieties mean that canon is (1) idempotent, (2) picks the same representative for the terms that are equal modulo T, (3) does not introduce new variables, and (4) recursively canonizes subterms. T is canonizable if it admits a canonizer.

A solver for T is a function (algorithm)

$$\texttt{solve} \ : \ \mathcal{T}_\Sigma(\mathcal{X}) \times \mathcal{T}_\Sigma(\mathcal{X}) \ \rightarrow \ \{\bot\} \cup (\mathcal{X} \times \mathcal{T}_\Sigma(\mathcal{X}))^*$$

that returns, for every equality $t_1 \approx t_2$:

- $\bot$, if the formula $t_1 \approx t_2$ is T-unsatisfiable
- a substitution $\{x_1 \mapsto v_1, \ \cdots, \ x_n \mapsto v_n\}$, where $x_1, \cdots, x_n$ are pairwise distinct variables occurring in $t_1 \approx t_2$ but not in $v_1, \cdots, v_n$, otherwise. Such a substitution have to comply with the following property:

  $$\models_{\mathrm{T}} t_1 \approx t_2 \ \longleftrightarrow \ \exists y_1, \cdots, y_m. \, (x_1 \approx v_1 \ \wedge \ \cdots \ \wedge \ x_n \approx v_n)$$

  where $y_1, \cdots, y_m$ are the variables occurring in $v_1, \cdots, v_n$.

The theory T is solvable if it admits a solver function. We say that T is a *Shostak theory* [111] if it is convex, canonizable and solvable. In the next section, we will give examples of such theories.

2.3 Some interesting theories in SMT

The SMT library [105] initiative provides a description for some interesting theories, such as functional arrays, rational and integer numbers. In this section, we present the axiomatization of these theories. We also provide the axioms of the free theory of equality, the theory of enumerated data types and the AC theory.

In the following, we assume given a signature $\Sigma = (\mathcal{F}, \mathcal{P}, \mathcal{S}, \tau_{\mathcal{F}}, \tau_{\mathcal{P}})$, a set of variables $\mathcal{X}$ disjoint from $\mathcal{F}$ and $\mathcal{P}$ and a mapping $\tau_{\mathcal{X}} : \mathcal{X} \to \mathcal{S}$.

2.3.1 The free theory of equality

The free theory of equality with uninterpreted symbols is certainly one of the most commonly used theories in SMT. It is defined by the set of *well-sorted sentences* obtained from the axioms schemes given in Figure 2.1. For each sort s, we have (1) the reflexivity axiom, (2) the symmetry axiom and (3) the transitivity axiom. Moreover, we have (4) the congruence property for each function symbol and (5) the congruence property for each predicate symbol.

$$
\mathcal{E} = \left\{ \begin{array}{lll}
1. & \forall x. \quad x \approx_s x & \text{for each } s \in \mathcal{S} \\
2. & \forall x, y. \quad x \approx_s y \;\Rightarrow\; y \approx_s x & \text{for each } s \in \mathcal{S} \\
3. & \forall x, y, z. \\
 & \qquad x \approx_s y \wedge y \approx_s z \;\Rightarrow\; x \approx_s z & \text{for each } s \in \mathcal{S} \\
4. & \forall x_1, \cdots, x_n, y_1, \cdots, y_n. & \left\{ \begin{array}{l} \text{for each } s \in \mathcal{S} \\ \text{for each } f \in \mathcal{F} \\ \tau_{\mathcal{F}}(f) = s_1 \times \cdots \times s_n \to s \end{array} \right. \\
 & \qquad \bigwedge_{i=1}^{n} x_i \approx_{s_i} y_i \;\Rightarrow \\
 & \qquad f(x_1, \cdots, x_n) \approx_s f(y_1, \cdots, y_n) & \\
5. & \forall x_1, \cdots, x_n, y_1, \cdots, y_n. & \left\{ \begin{array}{l} \text{for each } s \in \mathcal{S} \\ \text{for each } p \in \mathcal{P} \\ \tau_{\mathcal{P}}(p) = s_1 \times \cdots \times s_n \end{array} \right. \\
 & \qquad \bigwedge_{i=1}^{n} x_i \approx_{s_i} y_i \;\Rightarrow \\
 & \qquad (p(x_1, \cdots, x_n) \Leftrightarrow p(y_1, \cdots, y_n)) &
\end{array} \right.
$$

Figure 2.1: The axioms schemes defining the free theory of equality

The identity function is a suitable canonizer for this theory. However, it does not admit a solver function. This theory is widely used in software verification

to abstract the behavior of programs functions and sub-routines. The validity of quantifier-free conjunctions of literals modulo this theory can be decided using congruence closure algorithms or ground completion procedures [107, 97, 74, 8].

Example. Let Σ be a signature, such that $\mathcal{F} = \{a,\ b,\ c,\ f,\ g\}$, $\mathcal{P} = \emptyset$, $\mathcal{S} = \{s_1, s_2\}$, where $\tau_{\mathcal{F}}(a) = \tau_{\mathcal{F}}(b) = \tau_{\mathcal{F}}(c) = s_1$, $\tau_{\mathcal{F}}(f) = s_1 \to s_2$ and $\tau_{\mathcal{F}}(g) = s_1 \times s_2 \to s_1$. The following formula is valid, because its negation φ is unsatisfiable modulo the free theory of equality

$$\models_{\mathcal{E}} \quad \neg \underbrace{(b \approx_{s_1} a \ \wedge \ b \approx_{s_1} c \ \wedge \ \neg(g(a, f(a)) \approx_{s_1} g(c, f(b))))}_{\varphi}$$

In fact, the two first equalities of φ entail $f(a) \approx_{s_2} f(b)$ modulo $\mathcal{E}$. They also entail (with the intermediate fact) the equality $g(a, f(a)) \approx_{s_1} g(c, f(b))$ modulo $\mathcal{E}$. The formula φ is unsatisfiable, because there is no model that satisfies both $(g(a, f(a)) \approx_{s_1} g(c, f(b)))$ and its negation $\neg(g(a, f(a)) \approx_{s_1} g(c, f(b)))$.

2.3.2 Linear arithmetic

The theories of linear arithmetic over rationals and integers are defined over a signature Σ of the form:

- $\mathcal{F} = \{0,\ 1,\ +\}$
- $\mathcal{P} = \{\leq\}$
- $\mathcal{S} = \{s_{\text{LA}}\}$, where $s_{\text{LA}} \in \{\mathsf{rat},\ \mathsf{int},\ \mathsf{nat}\}$

where 0 and 1 are constants of sort s_{LA}, $+$ is a function symbol of sort $s_{\text{LA}} \times s_{\text{LA}} \to s_{\text{LA}}$, and $\leq$ is a predicate symbol of sort $s_{\text{LA}} \times s_{\text{LA}}$.

Linear rational arithmetic (LRA)

Linear rational arithmetic is defined over the sort $s_{\text{LA}} = \mathsf{rat}$. This theory is given by the set LRA of axioms, such that:

$$\text{LRA} \ = \ \mathcal{E} \ \cup \ \text{LRA}_{\text{S}} \ \cup \ \text{LRA}_{\text{NS}}$$

where $\mathcal{E}$ contains the axioms schemes given in Figure 2.1. The set $\mathrm{LRA_S}$ defines the equational part of linear rational arithmetic as follows:

$$\mathrm{LRA_S} = \begin{cases} 1. & \forall x, y, z.\ x + (y + z) \approx (x + (y + z)) & \text{associativity} \\ 2. & \forall x, y.\ x + y \approx y + x & \text{commutativity} \\ 3. & \forall x.\ x + 0 \approx x & \text{identity} \\ 4. & \forall x. \exists y.\ x + y \approx 0 & \text{inverse} \\ 5. & \forall x.\ \underbrace{x + \cdots + x}_{n \text{ times}} \approx 0 \Rightarrow x \approx 0 & \text{torsion-freeness} \\ 6. & \forall x. \exists y.\ \underbrace{y + \cdots + y}_{n \text{ times}} \approx x & \text{divisibility} \\ 7. & \neg\, 1 \approx 0 & \text{non-triviality} \end{cases}$$

The axioms $1,\ 2,\ 3$ and 4 are those of an abelian group. Axiom 4 allows to build negative rationals and axiom 6 allows to construct rationals numbers, such as $\frac{1}{2}$. Items 5 and 6 are axioms schemes that hold for every natural number $n > 0$. LRA is not finitely axiomatizable, because of these two axioms schemes. In practice, we use the abbreviation $n\,x$ to mean "x added to itself n times" (*i.e.* $x + \cdots + x$). Axiom 7 says that the group contains at least one non-zero element. The set $\mathrm{LRA_{NS}}$ contains the following axioms that define the total order $\leq$:

$$\mathrm{LRA_{NS}} = \begin{cases} 8. & \forall x, y.\ (x \leq y \wedge y \leq x) \Rightarrow x \approx y & \text{antisymmetry} \\ 9. & \forall x, y, z.\ (x \leq y \wedge y \leq z \Rightarrow x \leq z) & \text{transitivity} \\ 10. & \forall x, y.\ (x \leq y \vee y \leq x) & \text{totality} \\ 11. & \forall x, y, z.\ (x \leq y \Rightarrow x + z \leq y + z) & \text{ordered} \\ 12. & \neg\, 1 \leq 0 & \text{non-triviality} \end{cases}$$

LRA is convex, stably infinite and canonizable. For instance, a canonizer is obtained by viewing LRA-terms as sums $\sum_{i=1}^{n} a_i x_i + c$ of ordered monomials, where the x_i are pairwise distinct and each rational coefficient a_i is different from zero. This canonizer function simulates the axioms $1, 2, 3$ and 4 of the abelian group. Moreover, the equational part defined by $\mathrm{LRA_S}$ is solvable. A solver function is provided by the Gaussian elimination algorithm.

Linear integer arithmetic (LIA)

Linear integer arithmetic is defined over the sort $s_{\text{LA}} = \mathsf{int}$. This theory is usually presented as a construction above Presburger arithmetic [80]. The latter theory axiomatizes natural number and is given below.

The canonizer we defined for LRA is suitable for linear integer arithmetic. Moreover, this theory is stably infinite and the equational part is solvable [104, 66]. However, it is not convex. In fact, the formula below is valid modulo LIA

$$\models_{\text{LIA}} \quad \forall x.\ 0 \leq x \leq 1 \Rightarrow x \approx 0 \vee x \approx 1$$

but none of the formulas below are.

$$\models_{\text{LIA}} \quad \forall x.\ 0 \leq x \leq 1 \Rightarrow x \approx 0$$

and

$$\models_{\text{LIA}} \quad \forall x.\ 0 \leq x \leq 1 \Rightarrow x \approx 1$$

Presburger arithmetic (NAT)

Presburger arithmetic is defined over the sort $s_{\text{LA}} = \mathsf{nat}$. This theory is given by the set of axioms below, plus the axioms schemes of Figure 2.1 and the axioms defining the total order $\leq$.

$$\text{NAT} = \begin{cases} 1. & \forall x.\ \neg(x + 1 \approx x) & \text{zero} \\ 2. & \forall x.\ x + 0 \approx x & \text{plus zero} \\ 3. & \forall x, y.\ x + 1 \approx y + 1 \ \Rightarrow\ x \approx y & \text{successor} \\ 4. & \forall x, y.\ x + (y + 1) \approx (x + y) + 1 & \text{plus successor} \\ 5. & \left(\begin{array}{l} \varphi\,\{x \mapsto 0\} \quad \wedge \\ \forall x.\ \varphi \Rightarrow \varphi\,\{x \mapsto x + 1\} \end{array}\right) \Rightarrow \forall x.\, \varphi & \text{induction} \end{cases}$$

where "induction" is an axiom scheme that holds for every Σ-formula φ such that $\mathcal{F}vars(\varphi) = \{x\}$.

Deciding linear arithmetic

The main procedures used in state-of-the-art SMT solvers to decide quantifier-free conjunctions of literals modulo LRA are the simplex algorithm, the Fourier-Motzkin variables elimination method, or variants thereof [80, 110]. Moreover, LRA admits quantifiers elimination. In fact, Fourier-Motzkin's algorithm also decides LRA-satisfiability of quantified conjunctions of literals.

In order to decide quantifier-free conjunctions of literals modulo LIA, most of state-of-the-art SMT solvers implement extensions of the simplex algorithm, such as branch-and-bound, cutting-planes and branch-and-cut [110]. Extensions of the Fourier-Motzkin method, such as Omega-Test [104] and interval calculus are also used for this purpose. However, the Fourier-Motzkin algorithm does not scale in practice and saturates the memory even over rationals.

Example 1. *The following formula is valid modulo LRA.*

$$\models_{\text{LRA}} \quad \forall x, y.\ (\underbrace{x + y}_{t_1} \leq 0 \ \wedge \ \underbrace{-x + y}_{t_2} \leq 0 \ \wedge \ 0 \leq \underbrace{2y}_{t_3}) \ \Rightarrow \ x \approx 0$$

In fact, the validity can be shown as follows:

1. *we first deduce that* $\underbrace{2y}_{t_3 = t_1 + t_2} \leq 0$, *because* $t_1 + t_2 \leq t_2$ *and* $t_2 \leq 0$
2. *we then deduce* $t_3 \approx 0$, *because* $t_3 \leq 0$ *and* $0 \leq t_3$. *This implies* $y \approx 0$
3. *from 1 and 2, we deduce* $t_2 \approx 0$, *because* $t_3 = 0 \leq t_2$ *and* $t_2 \leq 0$, *and we conclude.*

Example 2. *The formula below is valid modulo LIA (resp. NAT), because its negation* φ *is unsatisfiable. However, this formula is not LRA-valid, because* φ *is LRA-satisfiable.*

$$\models_{\text{LIA}} \quad \neg\, (\underbrace{\exists x.\ x + x \approx 1}_{\varphi})$$

2.3.3 Extensional functional arrays

The theory of extensional functional arrays (ARR) is defined over a signature Σ of the form:

- $\mathcal{F} = \{\ get, set\ \}$
- $\mathcal{P} = \emptyset$
- $\mathcal{S} = \{sa, si, sv\}$

where get is a binary function symbol of sort $sa \times si \to sv$, set is a ternary function symbol of sort $sa \times sv \times si \to sa$. Intuitively, sa is the sort of arrays, si is the sort of indexes and sv is the sort of values stored in arrays cells. The symbol get is used to read the values of an array at given indexes and set allows to update the content of cells. More formally, this theory is defined by the set ARR of axioms given by

$$\text{ARR} \;=\; \mathcal{E} \;\cup\; \text{ARR}_{\text{NS}}$$

where $\mathcal{E}$ are the axioms schemes given in Figure 2.1 and ARR_{NS} is the set defined as follows:

$$\text{ARR}_{\text{NS}} = \begin{cases} 1. & \forall x_a, x_v, x_i. & get(set(x_a, x_v, x_i), x_i) \approx_{sv} x_v \\ 2. & \forall x_a, x_v, x_i, x_j. & x_i \approx_{si} x_j \;\vee\; get(set(x_a, x_v, x_i), x_j) \approx_{sv} get(x_a, x_j) \\ 3. & \forall x_a, x_b. & (x_a \approx_{sa} x_b) \;\vee\; \exists\, x_i. \neg\, (get(x_a, x_i) \approx_{sv} get(x_b, x_i)) \end{cases}$$

the first and the second axioms specify the meaning of get and set, as explained above. The third one is called the (weak) extensionality axiom. It specifies the meaning of the equality predicate $\approx_{sa}$.

The theory of functional arrays is very useful in software verification. It is used to specify memory models of programming languages and to abstract finite arrays data structures. Note that, this theory is not convex. In fact, the formula below is valid modulo ARR

$$\models_{\text{ARR}} \quad \begin{array}{l} \forall x_a, x_i, x_j, x_v, x_w. \\ \quad x_v \approx_{sv} get(set(x_a, x_w, x_i), x_j) \;\Rightarrow\; (x_v \approx_{sv} x_w \;\vee\; x_v \approx_{sv} get(x_a, x_j)) \end{array}$$

but none of the following formulas are.

$$\models_{\text{ARR}} \quad \begin{array}{l} \forall x_a, x_i, x_j, x_v, x_w. \\ \quad x_v \approx_{sv} get(set(x_a, x_w, x_i), x_j) \;\Rightarrow\; x_v \approx_{sv} x_w \end{array}$$

and

$$\models_{\text{ARR}} \quad \begin{array}{l} \forall x_a, x_i, x_j, x_v, x_w. \\ \quad x_v \approx_{sv} get(set(x_a, x_w, x_i), x_j) \;\Rightarrow\; x_v \approx_{sv} get(x_a, x_j) \end{array}$$

The quantifier-free fragment of ARR is decidable. During his thesis [95], Nelson has already considered the design of a decision procedure for the theory of arrays without extensionality. Recent algorithms [75, 64, 47, 24, 116] use various

techniques, such as the reduction of the problem to the free theory of equality, the use of axioms instantiation mechanisms to saturate the search space, the elimination of *read* or *write* operations, etc.

Example 3. *The following formula is valid modulo ARR.*

$$\models_{\mathrm{ARR}} \quad \begin{array}{l} \forall x_a, x_i, x_j, x_v, x_w. \\ \qquad get(x_a, x_j) \approx_{sv} x_v \;\; \Rightarrow \;\; get(set(x_a, x_i, x_v), x_j) \approx_{sv} x_v \end{array}$$

In fact, we conclude by axiom 1 if $x_i \approx_{si} x_j$. Otherwise, we conclude using the hypothesis $get(x_a, x_j) \approx_{sv} x_v$ by axiom 2.

2.3.4 Enumerated data types

In order to define a particular theory (*i.e.* an instance) of enumerated data types, one requires a signature Σ of the form:

- $\mathcal{F} = \{C_1, \; \cdots, \; C_n\}$, where n is a fixed natural number
- $\mathcal{P} = \emptyset$
- $\mathcal{S} = \{s\}$

where the function symbols C_i are constants of sort s called constructors. This theory is then defined by the set Enum of axioms, such that:

$$\mathrm{Enum} \;\; = \;\; \mathcal{E} \;\; \cup \;\; \mathrm{Enum_S} \;\; \cup \;\; \mathrm{Enum_{NS}}$$

where $\mathcal{E}$ is the set of axioms schemes given in Figure 2.1, the set $\mathrm{Enum_S}$ contains an axiom scheme saying that the constructors are pairwise distinct and $\mathrm{Enum_{NS}}$ has an axiom saying that a variable of sort s is necessarily equals to a constructor

$$\mathrm{Enum_S} \;\; = \left\{ \; 1. \quad \neg \left(C_i \approx_s C_j \right) \qquad \text{for each constants } C_i, C_j \text{ such that } i \neq j \right.$$

$$\mathrm{Enum_{NS}} = \left\{ \; 2. \quad \forall x. \quad x \approx_s C_1 \; \vee \; \cdots \; \vee \; x \approx_s C_n \right.$$

The identity function is a suitable canonizer for Enum. Moreover, we can easily define a solver for the part $\mathrm{Enum_S}$. Given an equality $t_1 \approx_s t_2$, such a solver returns:

- the substitution $\{s \mapsto t\}$ if s is a variable, or

- the substitution $\{t \mapsto s\}$ if t is a variable but not s, or
- $\emptyset$ if s and t are syntactically equal constructors, or
- $\bot$ if s and t are distinct constructors.

However, Enum is neither stably infinite nor convex as shown by the following example.

Example. Consider the theory RGB of enumerated data types defined over the signature Σ such that $\mathcal{F} = \{red, green, blue\}$, $\mathcal{P} = \{\approx_{rgb}\}$ and $\mathcal{S} = \{rgb\}$. Every Σ-structure that is a model for RGB has a finite cardinality because of axiom 2. Moreover, the following formula is valid modulo RGB

$$\models_{\text{RGB}} \quad \forall x. \quad x \approx_{rbg} red \ \vee \ x \approx_{rbg} green \ \vee \ x \approx_{rbg} blue$$

but none of the following formulas are.

$$\models_{\text{RGB}} \quad \forall x. \quad x \approx_{rbg} red$$

and

$$\models_{\text{RGB}} \quad \forall x. \quad x \approx_{rbg} green$$

and

$$\models_{\text{RGB}} \quad \forall x. \quad x \approx_{rbg} blue$$

2.3.5 The AC theory

Given a signature Σ, such that:

- $\mathcal{F} = \{f_1, \cdots, f_n\}$
- $\mathcal{P} = \emptyset$
- $\mathcal{S} = \{s_1, \cdots, s_m\}$

where each f_i is a function symbol of sort $s_j \times s_j \to s_j$, with $s_j \in \mathcal{S}$. The AC theory *over this signature* is defined by:

$$\text{AC} \ = \ \mathcal{E} \ \cup \ \text{AC}_{\text{NS}}$$

Again, $\mathcal{E}$ is the set of axioms schemes given in Figure 2.1, whereas $\mathrm{AC}_{\mathrm{NS}}$ is the set of axioms schemes defined below

$$\mathrm{AC}_{\mathrm{NS}} = \begin{cases} 1. \quad \forall x, y, z. \quad f_i(x, f_i(y, z)) \approx_{s_j} f_i(f_i(x, y), z) & \text{for each } f_i \in \mathcal{F} \\ & \text{of sort } s_j \times s_j \to s_j \\ 2. \quad \forall x, y. \quad f_i(x, y) \approx_{s_j} f_i(y, x) & \text{for each } f_i \in \mathcal{F} \\ & \text{of sort } s_j \times s_j \to s_j \end{cases}$$

AC is not a Shostak theory, because one cannot provide a solver function for arbitrary user-defined associative and commutative function symbols. For instance, one can easily build the substitution $\{a_1 \mapsto -2a_2\}$ from the equality $a_1 + 2a_2 = 0$. But, one cannot provide a substitution of the form $\{b_1 \mapsto \cdots\}$ or of the form $\{b_2 \mapsto \cdots\}$ for the equality $b_1 \cap b_2 = \emptyset$, where a_1, a_2, b_1 and b_2 are uninterpreted well-sorted constants.

However, the AC theory is canonizable. The canonizer defined in [70] is based on flattening and sorting techniques which simulate associativity and commutativity, respectively. We can then build sorted degenerate trees and use them as canonical form of terms, to avoid the use of function symbols with variable arities.

Given a signature Σ, where $\mathcal{F} = \mathcal{F}_{\mathrm{U}} \uplus \mathcal{F}_{\mathrm{AC}}$ is a disjoint union of uninterpreted and AC function symbols, and a total ordering $\trianglelefteq$ on terms, the canonizer $\mathtt{canon}_{AC}$ we described above can be defined as follows:

$$\begin{array}{lcll} \mathtt{canon}_{AC}(x) & = & x & \text{when } x \in \mathcal{X} \\ \mathtt{canon}_{AC}(f(\vec{v})) & = & f(\mathtt{canon}_{AC}(\vec{v})) & \text{when } f \in \Sigma_{\mathrm{U}} \\ \mathtt{canon}_{AC}(u(t_1, t_2)) & = & u(s_1, u(s_2, \ldots, u(s_{n-1}, s_n)\ldots)) & \text{when } u \in \Sigma_{\mathsf{AC}} \\ & & \text{where } t'_i = \mathtt{canon}_{AC}(t_i) \text{ for } i \in [1, 2] & \\ & & \text{and } \{\!\{s_1, \ldots, s_n\}\!\} = \mathcal{A}_{\{u\}}(t'_1) \cup \mathcal{A}_{\{u\}}(t'_2) & \\ & & \text{and } s_i \trianglelefteq s_{i+1} \text{ for } i \in \{1, \cdots, n-1\} & \end{array}$$

where $\mathcal{A}_{\{u\}}(t)$ is the multiset of the $\{u\}$-aliens of t. Note that, the function $\mathtt{canon}_{AC}$ is a canonizer for the union of the theories AC and $\mathcal{E}$, since it traverse uninterpreted functions (line 2). The lemma below states that $\mathtt{canon}_{AC}$ solves the word problem for the union of AC and $\mathcal{E}$. Its proof follows directly from the one given in [37].

Lemma 4. *Given two terms s and t over Σ, we have*

$$\models_{\mathcal{E},\mathrm{AC}} \; s \approx t \quad \Rightarrow \quad \mathtt{canon}_{AC}(s) = \mathtt{canon}_{AC}(t)$$

The use of a canonizer to handle the AC properties of function symbols is much more efficient than an axiomatic approach. In fact, to prove that

$$\models_{\mathrm{AC}} \quad u(c_1, u(c_2, \ldots, u(c_n, c_{n+1})\ldots) \approx u(u(\ldots u(c_{n+1}, c_n)\ldots, c_2), c_1)$$

is AC-valid when u is an AC symbol, the instantiation approach used in SMT solvers may explicitly introduce the $\frac{(2n)!}{n!}$ AC-equivalent terms in the context of the prover, while the function $\mathtt{canon}_{AC}$ simply flattens AC-terms, sorts them and builds degenerate trees.

Example 5. *Consider the following example, where $\cup$ is an AC function symbol.*

$$\models_{\mathrm{AC}} \quad \forall x_1, x_2, x_3, x_4. \quad (x_1 \cup x_4) \cup (x_3 \cup x_2) \approx (x_1 \cup (x_2 \cup (x_3 \cup x_4)))$$

The term $(x_1 \cup (x_2 \cup (x_3 \cup x_4)))$ is the canonical form of $(x_1 \cup x_4) \cup (x_3 \cup x_2)$ modulo AC (using lexicographic order for sorting). Consequently, this formula is AC-valid.

CHAPTER 3

On Deciding Quantifier-Free Linear Integer Arithmetic

Quantifier-free linear arithmetic over integers is a must-have in many domains. State-of-the-art SMT solvers use extensions of either the simplex algorithm [42, 14, 66, 45] or the Fourier-Motzkin method [19, 21] to decide this theory. However, simplex extensions are often drowned in a huge search space when they come to perform a case-split analysis. In addition, they are not complete *w.r.t.* the deduction of all implied equalities. Conversely, Fourier-Motzkin extensions do not scale in practice, because they potentially introduce a double exponential number of intermediate inequalities, which saturates the memory.

In this chapter, we describe a new procedure, called FM-simplex, for deciding conjunctions of quantifier-free linear integer arithmetic constraints. Our procedure interleaves an exhaustive search for a model with bounds inference. New bounds are computed by solving auxiliary rational linear optimization problems using the simplex algorithm. Intuitively, each auxiliary problem simulates a run of Fourier-Motzkin's procedure that would eliminate all the variables simultaneously.

In Section 3.1, we first recall some preliminaries and results about linear systems of constraints. In particular, we introduce the notion of constant positive linear combinations of affine forms. We then recall the Fourier-Motzkin procedure, linear optimization, and the simplex algorithm.

In Section 3.2, we first show that Fourier-Motzkin's algorithm can be extended to compute constant positive linear combinations of affine forms. Then, we explain how to cast this problem into a linear optimization problem, which can hence be solved by the simplex algorithm. After that, we characterize when the solution set described by a conjunction of integer constraints can effectively be bounded along some direction. If there is no such bound, we prove that the solution set contains infinitely many integer solutions.

In Section 3.3, we present our new decision procedure for QF-LIA. FM-simplex uses the techniques given in Section 3.2 to find bounds on the solution set. If there

are none, then the procedure stops since there are infinitely many integer solutions. Otherwise, it performs a case-split analysis along the bounded direction and calls itself recursively to solve simpler sub-problems.

In Section 3.4, we show that we can either use Omega-Test's equalities solver to handle equalities constraints, or Gaussian elimination with some additional checks.

In Section 3.5, we illustrate the use of our procedure through simple examples that show the different scenarios.

In Section 3.6, we describe the implementation of our decision procedure in a small SMT solver, called CTRL-ERGO. This solver relies on an OCaml version of MiniSAT and uses a new preprocessing phase to simplify LET-IN and IF-THEN-ELSE constructs.

In Section 3.7, we measure the performances of our implementation on the QF-LIA benchmark and compare it with some state-of-the-art SMT solvers. Finally, we overview related and future works in Section 3.8.

Note that, the ideas developed in this chapter have been published in [22].

3.1 Preliminaries

3.1.1 Linear systems of constraints

In this section, we recall the usual notations and definitions for linear systems of constraints. We also give some known mathematical results that will be used in the proofs of some new theorems.

If m and n are integers then $[|m, n|]$ denotes the interval of integers bounded by m and n. Matrices are denoted by upper case letters like A and column vectors by lower case letters like x. We use letters like a, b and c to denote vectors of constants. We denote by A^t the transpose of the matrix A and by Ax the matrix product of A by the vector x. Depending on the context, we use the symbol $+$ to denote the addition over vectors and matrices, respectively. If A is a $m \times n$ matrix then $a_{i,j}$ denotes the element of A at position (i, j), a_i denotes the i-th row vector of A of size n, and A_j the j-th column vector of A of size m. If x is a vector, x_i denotes its i-th coordinate. If x and y are n-vectors of the same ordered vector space then $x \geq y$ denotes the conjunction of constraints:

$$\forall i \in [|1, n|], \quad x_i \geq y_i$$

Affine form. An affine form on $\mathbb{Q}^n$ is a function $\psi : \mathbb{Q}^n \to \mathbb{Q}$ of shape $\psi = \phi + t_c$, where the $\phi : \mathbb{Q}^n \to \mathbb{Q}$ is a linear form and t_c is a translation of direction $c \in \mathbb{Q}^m$. For instance, the term $-2x + 3y - 5$ is an affine form. Its linear form is $-2x + 3y$ and its translation of direction is -5.

Closed convex. A closed convex $K \subset \mathbb{Q}^n$ is defined using a linear system of constraints over $\mathbb{Q}$ as follows:

$$K \;=\; \left\{ \; x \in \mathbb{Q}^n \qquad \text{such that} \qquad Ax + b \leq 0 \; \right\}$$

where $A \in \mathbb{Q}^{m \times n}$, $b \in \mathbb{Q}^m$. By definition, K is the convex polytope of the rational solutions of the linear system $Ax + b \leq 0$.

Given a convex K, we are interested in this chapter in determining whether $K \cap \mathbb{Z}^n$ is empty or not. In other words, we want to know whether the system $Ax + b \leq 0$ of constraints has integer solutions.

Let us denote by L_i the affine forms

$$L_i : \left| \begin{array}{ccc} \mathbb{Q}^n & \longrightarrow & \mathbb{Q} \\ x & \longmapsto & (Ax + b)_i \end{array} \right.$$

where $i \in [|1, m|]$. FM-simplex relies on the computation of constants positive linear combinations of affine form. This notion is defined as follows:

Definition 6 (Constant positive linear combination of affine forms)**.** *Let $(\psi_i)_{i \in [|1,k|]}$ be a family of affine forms on $\mathbb{Q}^n$. An affine form ψ on $\mathbb{Q}^n$ is a positive linear combination of the $(\psi_i)_{i \in [|1,k|]}$ if there exists $(\lambda_i)_{i \in [|1,k|]}$ a family of nonnegative scalars such that:*

$$\psi = \sum_{i=1}^{k} \lambda_i \psi_i \quad \textit{and} \quad \sum_{i=1}^{k} \lambda_i > 0$$

Moreover, if ψ is a constant affine form then it is called a constant positive linear combination of the $(\psi_i)_{i \in [|1,k|]}$.

We recall below the original formulation of Farkas' lemma [110, 59] on rationals:

Theorem 7 (Farkas' lemma)**.** *Given a matrix $A \in \mathbb{Q}^{m \times n}$ and a vector $c \in \mathbb{Q}^n$, then*

$$\exists x \in \mathbb{Q}^m, x \geq 0 \;\wedge\; Ax = c \quad \Leftrightarrow \quad \forall y \in \mathbb{Q}^n, y^t A \geq 0 \Rightarrow y^t c \geq 0$$

In the proofs, we will rather use the following equivalent formulation:

Theorem 8 (Theorem of alternatives)**.** *Let A be a matrix in $\mathbb{Q}^{m\times n}$ and b a vector in $\mathbb{Q}^m$. The system $Ax + b \leq 0$ has no solution if and only if there exists $\lambda \in \mathbb{Q}^m$ such that $\lambda \geq 0$ and $A^t\lambda = 0$ and $b^t\lambda > 0$.*

We also use the following reduction result on integer matrices [113].

Definition 9 (Smith Normal Form for Integers)**.** *A matrix $A \in \mathbb{Z}^{m\times n}$ is in Smith normal form if any non diagonal coefficient of A is zero and*

$$\forall i \in [|1, min(n, m)|], \quad a_{i,i} \text{ divides } a_{i+1,i+1}$$

For instance , the matrix A given below is in Smith normal form. In fact, for all $i, j \in [|1, 3|]$ such that $i \neq j$, we have $a_{i,j} = 0$. Moreover, $a_{1,1}$ divides $a_{2,2}$ and $a_{2,2}$ divides $a_{3,3}$.

$$A = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 32 \end{pmatrix}$$

Theorem 10. *Any matrix A in $\mathbb{Z}^{m\times n}$ is elementary equivalent to a matrix in Smith normal form. Otherwise said, there exists $U \in \mathbb{Z}^{m\times m}$ and $V \in \mathbb{Z}^{n\times n}$ two matrices invertible over $\mathbb{Z}$ and D a matrix in Smith normal form such that:*

$$UAV = D$$

3.1.2 The Fourier-Motzkin algorithm

Let $K := \{\, x \in \mathbb{Q}^n \mid Ax + b \leq 0 \,\}$ be a closed convex where $A \in \mathbb{Q}^{m\times n}$ and $b \in \mathbb{Q}^m$, and C the set of the affine forms $L_i : x \mapsto (Ax + b)_i$. In order to decide whether K is empty in the rationals or not, the Fourier-Motzkin algorithm proceeds by iteratively eliminating all the variables from the set of affine forms. More precisely, the iteration k of the procedure consists in:

1. choosing a variable x_k to eliminate,

2. partitioning the current set C_k of affine forms into a set $C^0_{x_k}$ without x_k and a set $C^+_{x_k}$ (resp. $C^-_{x_k}$) where x_k has positive (resp. negative) coefficients,

3. computing the set C_{k+1} of new affine forms:

$$C_{k+1} \ := \ C^0_{x_k} \ \cup \ \Pi(C^+_{x_k} \times C^-_{x_k})$$

where Π calculates a positive linear combination $L_{i,j} = \alpha_{i,j}L_i + \beta_{i,j}L_j$, where x_k has been eliminated for each $L_i \in C^+_{x_k}$ and $L_j \in C^-_{x_k}$. Notice that if either $C^+_{x_k}$ or $C^-_{x_k}$ is empty, then Π returns an empty set.

This iterative process terminates when all the variables are eliminated. It then returns a possibly empty set C_f of constant affine forms. We say that K is unsatisfiable in $\mathbb{Q}$ if there exists $c \in C_f$ such that $c > 0$. The example below illustrates a run of the Fourier-Motzkin algorithm.

Example 11. *Consider the following set of affine forms:*

$$C_1 : \left\{ \begin{array}{lcllcllcl} L_1 & = & 2x + y, & L_2 & = & -2x + 3y - 5, & L_3 & = & x + z + 1, \\ L_4 & = & x + 5y + z, & L_5 & = & -x - 4y + 3, & L_6 & = & 3x - 2y + 2 \end{array} \right\}$$

Eliminating z from C_1 is immediate since it only appears positively:

$$C_2 : \left\{ \begin{array}{lcllcllcl} L_1 & = & 2x + y, & L_2 & = & -2x + 3y - 5, & L_5 & = & -x - 4y + 3, \\ L_6 & = & 3x - 2y + 2 & & & & & & \end{array} \right\}$$

We eliminate the variable x and compute the set C_3 below using the combinations: $L_7 = L_1 + L_2$, $L_8 = L_1 + 2L_5$, $L_9 = 2L_6 + 3L_2$, $L_{10} = L_6 + 3L_5$

$$C_3 : \left\{ \begin{array}{lcllcllcl} L_7 & = & 4y - 5, & L_8 & = & -7y + 6, & L_9 & = & 5y - 11, \\ L_{10} & = & -14y + 11 & & & & & & \end{array} \right\}$$

Finally, the variable y is in turn eliminated thanks to the following combinations: $L_{11} = 7L_7 + 4L_8$, $L_{12} = 7L_7 + 2L_{10}$, $L_{13} = 7L_9 + 5L_8$, $L_{14} = 14L_9 + 5L_{10}$

The iterative process terminates and returns the set

$$C_4 : \{ \ L_{11} \ = \ -11, \quad L_{12} \ = \ -13, \quad L_{13} \ = \ -47, \quad L_{14} \ = \ -99 \ \ \}$$

We notice that $c \leq 0$, for all $c \in C_4$. Consequently, we conclude that the convex defined by the set $\{ \ L_1 \leq 0, \ L_2 \leq 0, \ L_3 \leq 0, \ L_4 \leq 0, \ L_5 \leq 0, \ L_6 \leq 0 \ \}$ of constraints is satisfiable in the rationals.

The Fourier-Motzkin algorithm only provides a semi-decision procedure over integers. For instance, the application of the process above on the example

$$-2x \leq 0 \ \wedge \ 2x - 1 \leq 0$$

yields the set $C_f = \{-1\}$ of constants affine forms. However, this conjunction is unsatisfiable over integers, because $2x \in [|0, 1|]$ has no integer solution.

3.1.3 Linear optimization and the simplex algorithm

Let $A \in \mathbb{Q}^{m \times n}$ be a matrix and $b \in \mathbb{Q}^m$, $c \in \mathbb{Q}^n$ two vectors. A linear optimization problem in *standard form* has the following shape:

$$\begin{array}{ll} maximize & c^t\, x \\ subject\ to & A\, x \ \leq\ b \quad \wedge \quad x \geq 0 \end{array} \qquad (P)$$

where $x \in \mathbb{Q}^n$. The affine form $c^t\, x$ is called the *objective function* of the problem. Note that, problems with equalities constraints, unconstrained variables and inequalities with greater-or-equal are straightforwardly convertible as linear optimization problems in standard forms [38].

We say that P is *unbounded* if $A\, x \leq b \wedge x \geq 0$ is satisfiable in $\mathbb{Q}$, but the affine form $c^t\, x$ has no upper bound. It is *infeasible*, if $A\, x \leq b \wedge x \geq 0$ is unsatisfiable in $\mathbb{Q}$. A vector $x_s \in \mathbb{Q}^n$ is *a solution* of P if it satisfies

$$x_s \geq 0 \ \wedge\ A\, x_s \ \leq\ b \ \wedge\ (\forall x'_s \in \mathbb{Q}^n.\ (x'_s \geq 0 \ \wedge\ A\, x'_s \ \leq\ b) \ \Rightarrow\ c^t\, x'_s \ \leq\ c^t\, x_s)$$

This means that

- x_s satisfies $A\, x_s \leq b \wedge x_s \geq 0$
- x_s maximizes the value of the objective function $c^t\, x_s$

The simplex algorithm takes as input a linear optimization problem in standard form. It returns an optimal solution when it exists, infeasible if the constraints have no solution in $\mathbb{Q}$ or unbounded otherwise. It consists of three main steps:

Step 1. First, the simplex constructs an equivalent problem in *slake form*. For that, a new vector $s \in Q^m$ is introduced and P is rewritten as follows:

$$\begin{array}{ll} maximize & c^t\, x \\ subject\ to & s \ =\ -\, A\, x + b \quad \wedge \\ & x \geq 0 \ \wedge\ s \geq 0 \end{array}$$

This is equivalent to the following formulation, if we expand the vectors c, b, x, s and the matrix A

$$\begin{array}{ll} maximize & c_1\, x_1 + \cdots + c_n\, x_n \\ subject\ to & s_1 \ =\ -\, a_{1,1}\, x_1 \ - \cdots \ - a_{1,n}\, x_n + b_1 \quad \wedge \\ & \vdots \\ & s_m \ =\ -\, a_{m,1}\, x_1 - \cdots - a_{m,n}\, x_n + b_m \quad \wedge \\ & \bigwedge_{i=1}^{n} x_i \geq 0 \ \wedge\ \bigwedge_{i=1}^{m} s_i \geq 0 \end{array}$$

The variables on the left-hand side of the equality symbol are called *basic* variables. Those on the right-hand side are *non-basic* variables.

The *basic solution* of the constraints in slake form is obtained by replacing non-basic variables with 0, which forces each basic variable s_i to be equal to b_i. This solution is *feasible* if b_i are non-negative. Note that, a basic feasible solution for the conjunction of constraints does not necessarily maximize the value of the objective function.

Pivoting. The next steps of the simplex are based on *pivoting*. Given a system of linear equalities in slake form where the basic variable s_i is

$$s_i \;=\; a_{i,1}\, x_1 \;+\; \cdots \;+\; \boxed{a_{i,j}\, x_j} \;+\; \cdots \;+\; a_{i,n}\, x_n \;+\; b_i \qquad (1)$$

If the coefficient $a_{i,j} \neq 0$, then the pivoting operation of s_i with x_j is done as follows:

1. first, the equality (1) is transformed as follows using Gaussian elimination

$$x_j \;=\; -\,\frac{a_{i,1}}{a_{i,j}}\, x_1 \;-\; \cdots \;-\; \boxed{\frac{-1}{a_{i,j}}\, x_i} \;-\; \cdots \;-\; \frac{a_{i,n}}{a_{i,j}}\, x_n \;-\; \frac{b_i}{a_{i,j}} \qquad (2)$$

2. second, the substitution of the form $\{x_j \mapsto \cdots\}$ obtained from (2) is applied both on the other equalities and on the objective function. This yields a triangular system where x_j (*resp.* x_i) becomes a basic (*resp.* a non-basic) variable.

Step 2. The second step of the algorithm consists in finding a basic feasible solution for the conjunction constraints:

1. the simplex checks if the basic solution of the current slake form is feasible
2. if so, the algorithm goes to **Step 3** and tries to maximize the objective function
3. otherwise, it computes an initial feasible solution as follows:
   (a) the algorithm constructs an auxiliary problem of the form:

$$P_{aux} \begin{cases} maximize & -x_0 \\ subject\ to & s \;=\; vec(x_0, m) \;-\; A\, x \;+\; b \;\;\wedge \\ & x \geq 0 \;\;\wedge\;\; s \geq 0 \;\;\wedge\;\; x_0 \geq 0 \end{cases}$$

   where $x_0 \in \mathbb{Q}$ and $vec(x_0, m)$ is the vector $(x_0,\ \cdots,\ x_0)^t$ of size m. Note that, P_{aux} is not unbounded, since the maximum of objective function is bounded by 0, as $x_0 \geq 0$.

(b) then, it performs a pivot between x_0 and the basic variable x_i for which $b_i = \texttt{min-comp}(b)$, where `min-comp` returns the minimal component of a given vector. Note that, the slake form we obtain after this pivot has a feasible solution. In fact, b_i is minimal and negative, and the coefficients of x_0 in P_{aux}'s equalities are positive ($+1$).

(c) after that, it solves the slake form obtained in (b) using **Step 3**.

(d) let P_{aux}^{end} be the final slake form of **Step 3** and $\nu : \{x_0, x_1, \ldots, x_n\} \to \mathbb{Q}$ be the substitution that maximizes $-x_0$.

(e) if $\nu(x_0) \neq 0$ then, the original optimization problem is infeasible

(f) if $\nu(x_0) = 0$ then,

i. the simplex removes the variable x_0 from the equalities of P_{aux}^{end}

ii. then, it restores the original objective function and normalizes it *w.r.t.* the system of linear equalities obtained from i.

iii. at this point, we obtain an equivalent slake form of the original problem, where some pivot operation have been made. Moreover, the basic solution of the current slake form is feasible. The simplex thus moves to **Step 3**

Step 3. The third step is dedicated to the maximization of the objective function. It works on problems in slake form having a feasible basic solution. For that, the simplex repeatedly does the following:

1. it chooses a non-basic variable x_j in the objective function which is associated with a positive coefficient $c_j > 0$
2. if such a variable does not exists then, the objective function is maximized by the current feasible basic solution
3. otherwise, it chooses a basic variable x_i in the system of linear equalities such that the coefficient $a_{i,j}$ associated with x_j is negative ($a_{i,j} < 0$) and $-\frac{b_i}{a_{i,j}}$ is minimal
4. if the coefficient $a_{i,j}$ does not exists then, the problem is unbounded
5. otherwise, perform a pivot operation between x_i and x_j

Note that, the choice of x_i and x_j in the description above have to comply with a pre-determined order on variables, in order to avoid cycling. Otherwise, the termination of the simplex is not ensured.

Example 12. *Let us apply the simplex algorithm on the following optimization problem, where* $x_1 \geq 0$ *and* $x_2 \geq 0$*. We use the lexicographic order* $\prec_{lex}$ *on variables*

$$\begin{array}{lll} maximize & 2x_1 - 3x_2 & \\ subject\ to & -2x_1 - x_2 \leq -2 & \wedge \\ & -x_1 - 2x_2 \leq 1 & \wedge \\ & 3x_1 - x_2 \leq 3 & \end{array}$$

- *We first construct the slake form as follows, where* $\bigwedge_{i=1}^{3} s_i \geq 0$

$$\begin{array}{lllll} maximize & & 2x_1 - 3x_2 & & \\ subject\ to & s_1 & = & 2x_1 + x_2 - 2 & \wedge \\ & s_2 & = & x_1 + 2x_2 + 1 & \wedge \\ & s_3 & = & x_2 - 3x_1 + 3 & \end{array}$$

- *The basic solution is not feasible. In fact, if* $x_1 = x_2 = 0$*, then* $s_1 = -2 < 0$*. Thus, we have to find a feasible basic solution by the solving the following auxiliary problem, where* $x_0 \geq 0$

$$\begin{array}{lllll} maximize & & -x_0 & & \\ subject\ to & s_1 & = & x_0 + 2x_1 + x_2 - 2 & \wedge \\ & s_2 & = & x_0 + x_1 + 2x_2 + 1 & \wedge \\ & s_3 & = & x_0 + x_2 - 3x_1 + 3 & \end{array}$$

- *Pivoting on* x_0 *in the first equality yields* $\{x_0 \mapsto s_1 - 2x_1 - x_2 + 2\}$*. Applying this substitution on the problem above produces*

$$\begin{array}{lllll} maximize & & -s_1 + 2x_1 + x_2 - 2 & & \\ subject\ to & x_0 & = & s_1 - 2x_1 - x_2 + 2 & \wedge \\ & s_2 & = & s_1 - x_1 + x_2 + 3 & \wedge \\ & s_3 & = & s_1 - 5x_1 + 5 & \end{array}$$

  Note that, the basic solution for this problem is now feasible

- *We use* ***Step 3*** *to maximize the objective function of this auxiliary problem. The simplex chooses the first equality and the variable* x_1 *as a first pivot. Indeed,* $x_1 \prec_{lex}$

x_2, the coefficient of x_1 in the objective function is $2 > 0$ and $-\frac{b_i}{a_{i,j}} = \frac{2}{2}$ is minimal. This pivot produces $\{x_1 \mapsto -\frac{1}{2}x_0 + \frac{1}{2}s_1 - \frac{1}{2}x_2 + 1\}$. Normalizing the problem above with this substitution returns

$$\begin{array}{lrcll} maximize & & -x_0 & & \\ subject\ to & x_1 & = & -\frac{1}{2}x_0 + \frac{1}{2}s_1 - \frac{1}{2}x_2 + 1 & \wedge \\ & s_2 & = & \frac{1}{2}x_0 + \frac{1}{2}s_1 + \frac{3}{2}x_2 + 2 & \wedge \\ & s_3 & = & \frac{5}{2}x_0 - \frac{3}{2}s_1 + \frac{5}{2}x_2 & \end{array}$$

- *At this point, **Step 3** returns, because the objective function is maximized (all its coefficients are non-positive). Furthermore, z is a non-basic variable. Thus, it is evaluated to 0 by the feasible basic solution that maximizes $-x_0$*

- *We go back to **Step 2**. The simplex algorithm removes the variable x_0 from the conjunction of equalities*

$$\begin{array}{rcll} x_1 & = & \frac{1}{2}s_1 - \frac{1}{2}x_2 + 1 & \wedge \\ s_2 & = & \frac{1}{2}s_1 + \frac{3}{2}x_2 + 2 & \wedge \\ s_3 & = & -\frac{3}{2}s_1 + \frac{5}{2}x_2 & \end{array}$$

  Then, it restores the original objective function $2x_1 - 3x_2$ and normalizes it w.r.t. the equalities above. This yields $s_1 - 4x_2 + 2$

- *After that, it moves to **Step 3** and tries to solve the initial problem. Note that, the basic solution of its current slake form (shown below) is feasible*

$$\begin{array}{lrcll} maximize & & s_1 - 4x_2 + 2 & & \\ subject\ to & x_1 & = & \frac{1}{2}s_1 - \frac{1}{2}x_2 + 1 & \wedge \\ & s_2 & = & \frac{1}{2}s_1 + \frac{3}{2}x_2 + 2 & \wedge \\ & s_3 & = & -\frac{3}{2}s_1 + \frac{5}{2}x_2 & \end{array}$$

- *The coefficient of s_1 in the objective function is positive. Moreover, $-\frac{b_i}{a_{i,j}}$ is maximal for the third equality and $a_{i,j} = -\frac{3}{2} < 0$. Pivoting on s_1 in this equality yields*

$\{s_1 \mapsto -\frac{2}{3}s_3 + \frac{5}{3}x_2\}$ and normalizing the problem with this substitution produces

$$\begin{array}{lrcll} maximize & & & -\frac{3}{2}s_3 - \frac{7}{3}x_2 + 2 & \\ subject\ to & x_1 & = & -\frac{1}{3}s_3 + \frac{1}{3}x_2 + 1 & \wedge \\ & s_2 & = & -\frac{1}{3}s_3 + \frac{7}{3}x_2 + 2 & \wedge \\ & s_1 & = & -\frac{2}{3}s_3 + \frac{5}{3}x_2 & \end{array}$$

- *The objective function is now maximized, because all its coefficients are non-positive. Its maximal value is 2. It is reached when $x_1 = 1$ and $x_2 = 0$*

Duality in linear optimization

Given a linear optimization problem. It is sometimes more efficient to solve its *dual* problem. The notion of duality in linear optimization is introduced by the definition below:

Definition 13. *Let $A \in \mathbb{Q}^{m \times n}$ be a matrix and $b \in \mathbb{Q}^m$, $c \in \mathbb{Q}^n$ two vectors. The dual of the following optimization problem*

$$\begin{array}{ll} maximize & c^t\, x \\ subject\ to & A\, x \ \leq\ b \quad \wedge \quad x \geq 0 \end{array}$$

is defined by

$$\begin{array}{ll} maximize & b^t\, y \\ subject\ to & A^t\, y \ \geq\ c \quad \wedge \quad y \geq 0 \end{array}$$

where $x \in \mathbb{Q}^n$, $y \in \mathbb{Q}^m$

The theorem below enunciates the relationship between a linear optimization problem and its dual.

Theorem 14. *Let P be an optimization problem and Q its dual problem.*

- *If P is infeasible (resp. unbounded) then Q is unbounded (resp. infeasible).*
- *If P has a solution, let P^{max} be an equivalent slake form of P in which that basic solution is feasible and maximizes the objective function. Then, we can extract a solution for Q from P^{max}.*

Deciding QF-LRA with the simplex algorithm

Let $K := \{ x \in \mathbb{Q}^n \mid Ax + b \leq 0 \}$ be a closed convex where $A \in \mathbb{Q}^{m \times n}$ and $b \in \mathbb{Q}^m$. In order to decide whether K is empty, it suffices solve the following optimization problem

$$\begin{aligned} &maximize \quad 0 \\ &subject\ to \quad A\,(y - z) \ \leq \ b \quad \wedge \quad y \geq 0 \quad \wedge \quad z \geq 0 \end{aligned}$$

where $y, z \in \mathbb{Q}^n$ and $x = y - z$. This problem is not unbounded. If the simplex returns infeasible then K is empty. Otherwise, it returns a valuation for y and z. In this case, we can exhibit a model $x = y - z$ for K.

In general, when the simplex algorithm exhibits a rational model for a given problem, we cannot conclude that this problem in satisfiable in the integers, because the valuation of all the variables are not necessarily in $\mathbb{Z}$. Consequently, the simplex algorithm only provides a semi-decision procedure for QF-LIA.

3.2 Constant positive linear combinations of affine forms

In this section, we design an efficient technique for computing constant positive linear combinations of affine forms. We build an oracle which takes as input a set of affine forms (L_i) (or equivalently, a matrix A and a vector b) and meets the following specifications:

1. if there is no constant positive linear combination of the (L_i), it says so;
2. otherwise, it returns such a combination $\sum_i \lambda_i L_i$.

We first present a method based on the Fourier-Motzkin procedure. Then, we describe an efficient implementation based on the simplex algorithm and prove its soundness, completeness and termination.

3.2.1 Computing the combinations using Fourier-Motzkin

Let $K := \{ x \in \mathbb{Q}^n \mid Ax + b \leq 0 \}$ be a closed convex where $A \in \mathbb{Q}^{m \times n}$ and $b \in \mathbb{Q}^m$, and C the set of the affine forms $L_i : x \mapsto (Ax + b)_i$. Let C_f be the set of constants affine forms obtained by running the Fourier-Motzkin algorithm with the input C.

For every constant $c \in C_f$, we can easily retrieve a constant positive linear combination of the initial affine forms of the shape $\sum_i \lambda_i L_i$ that is equal to c, where λ_i are rational scalars. For that, we recursively unfold the definitional equalities $L_{i,j} = \alpha_{i,j} L_i + \beta_{i,j} L_j$ previously computed by Π in Fourier-Motzkin's execution.

Using the constant positive linear combinations of the form $c = \sum \lambda_i \, L_i$, we can refine the bounds of some initial affine forms as follows: since for any vector $x \in K$ and for any index j we have $L_j(x) \leq 0$, we deduce that $c = \sum \lambda_i \, L_i(x) \leq \lambda_j L_j(x)$, and we obtain a lower bound $\frac{c}{\lambda_j}$ on L_j as soon as $\lambda_j \neq 0$.

Let us reuse the results of example 11 to illustrate this mechanism:

Example 15. *Unfolding the equalities introduced by* Π *in example 11 yields the following constant positive linear combinations*

$$\begin{cases} -11 = L_{11} = 7L_7 + 4L_8 = \cdots = 11L_1 + 7L_2 + 8L_5 \\ -13 = L_{12} = 7L_7 + 2L_{10} = \cdots = 7L_1 + 7L_2 + 6L_5 + 2L_6 \\ -47 = L_{13} = 7L_9 + 5L_8 = \cdots = 5L_1 + 21L_2 + 10L_5 + 14L_6 \\ -99 = L_{14} = 14L_9 + 5L_{10} = \cdots = 42L_2 + 15L_5 + 33L_6 \end{cases}$$

Using the linear combination $11L_1 + 7L_2 + 8L_5 = -11$*, we can make the following deductions in the rationals:*

$$-1 \leq L_1 \quad , \qquad -\frac{11}{7} \leq L_2 \quad , \qquad -\frac{11}{8} \leq L_3$$

Furthermore, these deductions are refined as follows in the integers:

$$-1 \leq L_1 \quad , \qquad \left\lceil -\frac{11}{7} \right\rceil = -1 \leq L_2 \quad , \qquad \left\lceil -\frac{11}{8} \right\rceil = -1 \leq L_3$$

We deduce that every integer solution of the constraints defining K *satisfies the bounds:*

$$-1 \leq L_1 \leq 0 \quad , \qquad -1 \leq L_2 \leq 0 \quad , \qquad -1 \leq L_3 \leq 0$$

3.2.2 Computing the combinations using a simplex

While the Fourier-Motzkin algorithm can be used to compute all the relevant constant positive linear combinations of affine forms (modulo a multiplication by a positive rational constant), it does not scale in practice. In the following, we describe an efficient simplex-based alternative and show its soundness, completeness and termination. As opposed to Fourier-Motzkin, this new approach only attempts to compute one particular constant positive linear combination. An heuristic allows us to hopefully calculate the *most interesting* one.

Let $K := \{ \, x \in \mathbb{Q}^n \mid Ax + b \leq 0 \, \}$ be a closed convex where $A \in \mathbb{Q}^{m \times n}$ and $b \in \mathbb{Q}^m$. Let C be the set of m affine forms $L_i : x \mapsto (Ax + b)_i = \sum_{j=1}^{n} a_{i,j} \, x_j + b_i$.

Consider the linear combination $\sum_{i=1}^{m} \lambda_i \, L_i$ of these forms where $\lambda_1, \ldots, \lambda_m$ are rational scalars. This combination can be unfolded as follows:

$$\lambda_1 \, (\sum_{j=1}^{n} a_{1,j} \, x_j + b_1) \quad + \quad \cdots \quad + \quad \lambda_m \, (\sum_{j=1}^{n} a_{m,j} \, x_j + b_m)$$

Factorizing the x_i yields the following equivalent sum:

$$x_1 \, (\sum_{i=1}^{m} a_{i,1} \, \lambda_i) \quad + \quad \cdots \quad + \quad x_n \, (\sum_{i=1}^{m} a_{i,n} \, \lambda_i) \quad + \quad \sum_{i=1}^{m} b_i \, \lambda_i$$

Since we are only interested in computing *constant* positive linear combinations, we eliminate each variable x_k for $k \in [|1, m|]$ by requiring that $\sum_{i=1}^{m} a_{i,k} \, \lambda_i = 0$. Moreover, we heuristically look for a constant positive linear combination that maximizes the value of the final constant $\sum_{i=1}^{m} b_i \, \lambda_i$ in order to hopefully improve efficiency as this will be described in Section 3.3.

More formally, we try to compute such a constant positive linear combination by solving the following linear optimization problem in the rationals:

$$\begin{array}{lll} \mathit{maximize} & \sum_{i=1}^{m} b_i \, \lambda_i & \\ \mathit{subject\ to} & A^t \lambda \ = \ 0 & \wedge \\ & \sum_{i=1}^{m} \lambda_i \ > \ 0 & \wedge \\ & \bigwedge_{i=1}^{m} \lambda_i \ \geq \ 0 & \end{array}$$

This problem reminds of the dual simplex input, but here we have equalities $A^t \lambda = 0$ instead of the usual inequalities and an extra constraint $\sum_{i=1}^{m} \lambda_i > 0$ that discards the trivial solution assigning zero to each parameter λ_i.

In order to solve the optimization problem above, we first introduce a slack variable $s \in \mathbb{Q}$ and a positive parameter ε to transform the strict inequality

$$\sum_{i=1}^{m} \lambda_i > 0$$

into

$$\sum_{i=1}^{m} \lambda_i - \varepsilon = s \ \wedge \ s \geq 0$$

following Lemma 1 of [55]. Then, for sake of efficiency, we do not transform the equalities into conjunctions of inequalities as done in the literature. We rather solve them, modulo the constraints $\bigwedge_{i=1}^{m} \lambda_i \geq 0 \wedge s \geq 0$. This returns unsat if this system is inconsistent in $\mathbb{Q}$ modulo these non-negativeness constraints, or a conjunction

of equalities in slake form as shown below, where the variables λ_{b_i} (*resp.* λ_{n_k}) are basic (*resp.* non-basic).

$$\begin{cases} \lambda_{b_1} = c_{1.1}\,\lambda_{n_1} + c_{1.2}\,\lambda_{n_2} + \cdots + (d_1 + e_1\,\varepsilon) \\ \lambda_{b_2} = c_{2.1}\,\lambda_{n_1} + c_{2.2}\,\lambda_{n_2} + \cdots + (d_2 + e_2\,\varepsilon) \\ \vdots \qquad\qquad \vdots \qquad\qquad \vdots \end{cases}$$

After that, we initialize the simplex with this conjunction and try to maximize the objective function. The algorithm returns unsat if the given system has no solution, or unbound if the objective function has no upper bound, or a maximum m and a valuation (substitution) ν for the vector λ.

If the simplex algorithm returns unsat, then the oracle answers that there is no constant positive linear combination. If it returns unbound, the oracle just returns a positive constant. The value of this constant is irrelevant for the deductions made by the decision procedure in Section 3.3, as explained in Lemma 18 of Section 3.2.3 below. Otherwise, the simplex necessarily returns a solution with a non-positive maximum for the objective function. Indeed, if the maximum were to be positive, one could multiply coordinate-wise any solution λ by a constant larger than 1 and obtain another solution with a larger objective value. The oracle then returns the corresponding linear combination. Note that, as soon as the simplex exploration discovers a positive value for the objective function, the answer will eventually be unbound so it can exit immediately.

Soundness, completeness and termination

On top of the simplex algorithm we only added some substitutions, so the termination of this oracle follows directly from the one of the simplex algorithm.

We now justify that the introduction of the parameter ε does affect neither the soundness nor the completeness of the oracle. Let us denote by $P_{>0}$ the original problem and by $P_{\geq\varepsilon}$ the problem we send to the simplex algorithm. Remember that for both problems, the answer is either unsat or unbound or a solution with a non-positive evaluation of the objective function: indeed, if ν is a solution, so is $\alpha\nu$ for any scalar α with the constraint $\alpha > 0$ for $P_{>0}$, and $\alpha > 1$ for $P_{\geq\varepsilon}$, and the value of the objective function is multiplied accordingly. Moreover, any solution of $P_{\geq\varepsilon}$ is obviously a solution of $P_{>0}$. Let us now proceed with the proof by case analysis.

1. If $P_{>0}$ is unsat, so is $P_{\geq\varepsilon}$ by inclusion of solutions.

2. If $P_{\geq\varepsilon}$ is unsat, let us assume by contradiction that $P_{>0}$ is not unsat, hence has a solution ν. Then $\frac{\varepsilon}{\sum\lambda_i}\ \nu$ is a solution of $P_{\geq\varepsilon}$, a contradiction.

3. If $P_{\geq\varepsilon}$ is unbound, so is $P_{>0}$ by inclusion of solutions.

4. If $P_{>0}$ is unbound, we show that $P_{\geq\varepsilon}$ is also unbound. Let M be an arbitrary large value. By hypothesis on $P_{>0}$, there is a solution ν of $P_{>0}$ with an evaluation of the objective function greater than M. Then, if $\sum\lambda_i \geq \varepsilon$, ν is a solution of $P_{\geq\varepsilon}$ with an evaluation of the objective function greater than M. Otherwise, $\frac{\varepsilon}{\sum\lambda_i}\nu$ is a solution of $P_{\geq\varepsilon}$, with an evaluation of the objective function greater than $\frac{\varepsilon}{\sum\lambda_i}M$, hence greater that M since $\varepsilon > \sum\lambda_i$.

5. If $P_{>0}$ (resp. $P_{\geq\varepsilon}$) has a solution with a non-positive objective function, so has $P_{\geq\varepsilon}$ (resp. $P_{>0}$), since the other cases are impossible as shown above.

3.2.3 Convex polytopes with an infinite number of integer points

In this section, we prove that if there is no constant positive linear combination of a given set of affine forms (L_i), then the convex K defined by these forms has infinitely many integer solutions. This intermediate result is crucial for the soundness of the decision procedure presented in Section 3.3.

We equip $\mathbb{Q}^n$ with the usual scalar product associated with its canonical basis. We measure distances using the *supremum* norm $\|\cdot\|_\infty$. We use $B_\infty(x,r)$ to denote the closed ball centered in x and of radius r for that norm.

Theorem 16. *If there is no constant positive linear combination of the affine forms (L_i), then for all $R \in \mathbb{Q}^+$, there exists $w \in \mathbb{Q}^n$ such that:*

$$\textit{The convex } K \textit{ contains the ball } B_\infty(w,R)$$

Proof. Assume that there is no constant positive linear combination of the affine forms (L_i). Let $R \in \mathbb{Q}^+$. We define a vector $\gamma \in \mathbb{Q}^m$ as follows:

$$\forall i \in [|1,m|].\quad \gamma_i \;=\; R\,\|a_i\|_1 \;=\; R\sum_{j=1}^{n}|a_{i,j}|$$

Consider the new convex K' defined by:

$$K' \;:=\; \{\, x \in \mathbb{Q}^n \mid Ax + \gamma + b \leq 0 \,\}$$

Suppose for contradiction that K' is empty. Hence by Theorem 8, there exists a vector $\lambda \in \mathbb{Q}^m$ such that:

$$A^t\lambda = 0$$

But, this implies that the linear combination $\lambda^t(Ax+b) = \sum_i \lambda_i L_i$ is a constant, which contradicts the theorem's hypothesis. Therefore K' is not empty and contains a vector $w \in \mathbb{Q}^n$ such that:

$$Aw + b + \gamma \leq 0$$

Now, we prove that $B_\infty(w, R) \subseteq K$. Let $u \in \mathbb{Q}^n$ be a point in the ball. We have $\|u\|_\infty \leq R$, and by triangular inequality we obtain:

$$\forall i \in [|1, \ldots, m|] \quad (Au)_i \;\leq\; |(Au)_i| \;\leq\; \|a_i\|_1 \, \|u\|_\infty \;\leq\; R \, \|a_i\|_1 \;=\; \gamma_i$$

hence

$$A(w+u)+b \;=\; Aw + b + Au \;\leq\; Aw + b + \gamma \;\leq\; 0$$

This proves that $w + u$ belongs to the convex K. Thus, $B_\infty(w, R) \subseteq K$. □

Corollary 17. *If there is no constant positive linear combination of the (L_i) then $K \cap \mathbb{Z}^n$ contains infinitely many points, for $n > 0$.*

Proof. For any $N \in \mathbb{N}$, for any $x \in \mathbb{Q}^n$, the ball $B_\infty(x, N)$ contains at least $(2N)^n$ points with integer coordinates. □

We now provide an algorithm that computes an integer solution for the system of constraints defining the convex K, when there is no constant positive linear combination. This algorithm relies on the proof of Theorem 16.

Computing an integer solution

- Let $R = \frac{1}{2}$. We define $\gamma \in \mathbb{Q}^m$ as follows: $\forall i \in [|1, m|]. \;\; \gamma_i = \frac{1}{2} \sum_{j=1}^n |a_{i,j}|$,
- Let $w \in \mathbb{Q}^n$ be a rational solution of the system $Aw + b + \gamma \leq 0$. This solution can for instance be computed using a rational simplex,
- We define $\widehat{w} \in \mathbb{Z}^n$ as follows: $\forall i \in [|1, n|]. \;\; \widehat{w}_i = \texttt{round}(w_i)$, where `round` is the usual rounding operator over rationals.
- By construction, we have $\widehat{w} \in B_\infty(w, \frac{1}{2})$ because $|\widehat{w}_i - w_i| \leq \frac{1}{2}$, for all i in $[|1, n|]$. Thus, the integer point $\widehat{w}$ belongs to the convex K.

When there exists a constant positive linear combination of the affine forms (L_i), the following lemma will be used in our procedure:

Lemma 18. *If $c = \sum_i \lambda_i L_i$ is a constant positive linear combination of the (L_i), then*

- *if c is positive, then K is empty*
- *otherwise for every k such that $\lambda_k \neq 0$, and for any $x \in K$, $L_k(x)$ is bounded by:*

$$\frac{c}{\lambda_k} \leq L_k(x) \leq 0$$

Proof. For any vector $x \in K$, we have by definition of K and non-negativeness of λ_i that $\lambda_i L_i(x) \leq 0$. Hence $\sum_i \lambda_i L_i(x) \leq 0$, which concludes the first case. Since $\sum_i \lambda_i L_i = c$, then for all k such that $\lambda_k \neq 0$ and for any $x \in K$, we have $c - \lambda_k L_k(x) = \sum_{i \neq k} \lambda_i L_i(x) \leq 0$, which concludes the second case. Notice that, if the constant c is zero, then all the inequalities $L_k(x) \leq 0$ associated with a nonzero λ_k are in fact equalities. □

3.3 A new decision procedure for QF-LIA

In this section, we build a decision procedure for quantifier-free linear integer arithmetic based upon an oracle that fulfills the interface given at the beginning of Section 3.2.

3.3.1 The main algorithm

Let $A \in \mathbb{Q}^{m \times n}$ and $b \in \mathbb{Q}^m$. FM-simplex shall decide whether the system $Ax+b \leq 0$ of linear inequalities has a solution $x \in \mathbb{Z}^n$. Let $C = (L_i)_{i \in [|1,m|]}$ be the associated family of affine forms. Figure 3.1 sketches the algorithm. It takes as input both the system C of linear inequalities and an additional argument Eq representing affine relations between variables, *e.g.* a set of equalities, or a substitution, etc. This last argument is initially empty. The result of the decision procedure is stored in the $sols$ variable. It is a finite set of integer solutions, possibly empty, or an indeterminate infinite set of integer solutions. The $check$ functions at lines 6 and 10 compute the integer solutions of the given system and are intentionally kept abstract in the procedure. The special shape of the systems they deal with are detailed in Section 3.4.

The algorithm is recursive. Recursive calls are performed on smaller and smaller systems C until complete resolution. Branching is caused by the loop on line 15.

1. **global** $sols \leftarrow \emptyset$
2. **procedure LIA** $(C = (L_i), Eq)$
3. remove trivial inequalities $c \leq 0$ with c constant from C
4. **if** some c was positive then **return**
5. **if** $C = \emptyset$ **then**
6. $sols \leftarrow sols \cup check_1(Eq)$
7. **return**
8. **call** $oracle(C)$
9. **if** there is no constant positive linear combination **then**
10. $sols \leftarrow sols \cup check_\infty(Eq, C)$
11. **return**
12. let $\sum \lambda_i L_i = c$ be the linear combination found by the oracle
13. **if** $c > 0$ **then return**
14. choose k and $\mu > 0$ *s.t.* $\lambda_k \neq 0$ and μL_k has integer coefficients only
15. **for** all v from $\lceil \mu \frac{c}{\lambda_k} \rceil$ to 0 **do**
16. create a substitution σ from $\mu L_k(x_1, \ldots, x_n) = v$
17. **if** there is no possible substitution **then continue** to next iteration
18. remove L_k from C
19. apply σ to C
20. **call LIA**$(C, Eq \cup \{\sigma\})$
21. **return**
22. **end-procedure**

Figure 3.1: The main steps of the decision procedure

The results are merged along the various branches at lines 6 and 10. One can also consider that there are implicit statements $sols \leftarrow sols \cup \emptyset$ at lines 4, 13, and 17. Note that the algorithm performs computations only when going from the root to the leaves of the call tree. For the sake of clarity, we have described a simple version of the algorithm. An actual implementation would likely be more complicated. For instance, it would exit as soon as a branch finds an infinity of solutions or even a single solution if one is only interested in satisfiability. It could also use *splitting on demand* [17] at line 15.

From lines 3 to 7, the algorithm deals with degenerate systems that contain no inequalities or only trivial inequalities. At line 8, it calls the oracle described in Section 3.2 on C. It then does the following depending on the oracle's answer:

1. If the oracle answers that no suitable combination of the affine forms exists, Corollary 17 applies: there are infinitely many solutions with integer coordinates, assuming Eq imposes no restriction (hence the call to $check_\infty$). The decision procedure is done in this branch.

2. Otherwise, if the oracle returns a constant positive linear combination $\sum_i \lambda_i L_i$ that is equal to a positive constant, the first case of Lemma 18 applies: the system has no solution.

3. Otherwise, the oracle returns a constant positive linear combination $\sum_i \lambda_i L_i$ that is equal to a non-positive constant c. The second case of Lemma 18 can then be applied to infer lower bounds for some affine forms in C. Indeed, we have $\frac{c}{\lambda_k} \leq L_k \leq 0$ for all k such that $\lambda_k \neq 0$. The decision procedure chooses a value for k. Since the coefficients of μL_k are in $\mathbb{Z}$, for any point $x \in K \cap \mathbb{Z}^n$, $\mu L_k(x)$ is an integer between $\mu\frac{c}{\lambda_k}$ and 0. For each integer $v \in [|\mu\frac{c}{\lambda_k}, 0|]$, the procedure considers the equality $\mu L_k = v$ from which it infers a substitution if possible, applies the result to all the other affine forms in C and removes L_k from the system while updating Eq with the substitution. A recursive call to the procedure is then performed. If no solution is found after a complete exploration of all the possible integer values in $[|\mu\frac{c}{\lambda_k}, 0|]$, then the procedure returns at line 21 without updating $sols$. This is what happens if C has rational solutions but no integer solutions. Notice that, if the constant c is zero, then several implied equalities might appear at once. An optimized procedure should therefore compute a substitution taking all of them into account, rather than one after the other, as is done in Figure 3.1.

Producing explanations. When developing a decision procedure for an SMT solver, it is important to provide the most precise explanations in order to improve the backtrack level when branching. In our setting, the explanation of each lower bound $\frac{c}{\lambda_k}$ inferred by the procedure is the explanations of the inequalities $L_i \leq 0$ such that $\lambda_i \neq 0$. If a constraint has not participated in the inference process, its explanation is discarded.

3.3.2 Soundness, completeness and termination

In this section, we assume that the oracle and the equalities handling mechanism are sound, complete and terminating.

Termination of FM-simplex is obvious. Indeed, at each recursive call, one affine form at least is removed from the system.

Soundness depends on the completeness of the oracle: if it does not find any constant positive linear combination, there should be none. Theorems given in Section 3.2.3 then cover all the possible cases. Completeness comes from termination and soundness.

While the oracle can return any constant positive linear combination, it should strive to find the greatest one, for efficiency reasons: a positive constant if possible, and zero if not. This would — heuristically — prevent the algorithm from branching too early.

3.4 Handling equality constraints

In this section, we describe two different techniques for handling equalities in our decision procedure. The first technique is based on integer substitutions with slack variables. The second one uses the Gaussian elimination algorithm. We also give more details on the computations performed at the leaves of the call tree by the $check$ functions.

3.4.1 Integer substitutions with slack variables

We first consider the case where the substitution scheme introduces new slack variables: some variables $x_1, \ldots, x_i$ of the affine form L_k are expressed as affine combinations of the other variables $x_{i+1}, \ldots, x_n$ and of additional fresh variables $x_{n+1}, \ldots, x_{n+\ell}$ such that the integer solutions of $L_k(x_1, \ldots, x_n) = v$ are completely parameterized by $x_{i+1}, \ldots, x_n, x_{n+1}, \ldots, x_{n+\ell}$. Removing L_k from C, applying the substitution on $(L_i)_{i \neq k}$ and updating Eq with σ produces a system equisatisfiable to C when $L_k(x_1, \ldots, x_n)$ is evaluated to v.

A possible technique to compute integer substitutions at line 16 is the well known Generalized GCD test [10] with the approach given by Pugh in the Omega-Test procedure [104]. Note that the substitution may have only one solution, *e.g.* $2x = 6$. The substitution may also not exist, *e.g.* $5x = 2$, in which case, the exit case described line 17 applies. In this scenario, the functions $check$ are implemented as follows:

- For $check_1$, the solutions are purely constrained by Eq. More precisely, the set of integer solutions is parameterized by the set of variables that are never the target of substitution in Eq. Moreover, the Eq system has been built only from adding successively (cf. line 20) new substitutions not featuring the variables previously substituted (cf. line 19), so it is never inconsistent. Therefore, only two situations are possible when Eq is passed to $check_1$: either Eq involves all the variables of the system, in which case there is exactly one solution, or it does not (some variables have not been substituted) and there are infinitely many integer solutions.

- For $check_\infty$, the solutions are constrained by both Eq and the remaining affine forms in C for which there is no constant positive linear combination (cf. line 9). However, since Eq is never inconsistent and the substitutions it contains are progressively applied on C at line 19, explicit solutions can easily be computed thanks to the algorithm described in 3.2.3.

3.4.2 Intersection with an affine subspace

Before we present the second scenario, we introduce and prove, in this section, an intermediate result related to the use of Gaussian elimination to handle equalities in our decision procedure. In addition to the convex K defined by the system $Ax + b \leq 0$, we consider another convex $K' \subset \mathbb{Q}^n$ defined by ℓ equations

$$K' = \left\{ x \in \mathbb{Q}^n \qquad \text{such that} \qquad A'x + b' = 0 \right\}$$

where $A' \in \mathbb{Z}^{\ell \times n}$, $b' \in \mathbb{Z}^\ell$. We prove here a sufficient condition for the intersection $K \cap K' \cap \mathbb{Z}^n$ to be infinite when $K \cap \mathbb{Z}^n$ is known to be infinite.

Let $(e_1, \ldots, e_n)$ be the canonical basis of $\mathbb{Q}^n$. We suppose that there exists $i_1, \ldots, i_j$ such that K is invariant by any translation of direction e_{i_k} for $k \in [|1, j|]$. Hence, if we define $E := \langle e_{i_1}, \ldots, e_{i_j} \rangle$ to be the vector space generated by these vectors, the convex K is invariant by any translation of direction $e \in E$. Let us denote by $\pi : \mathbb{Q}^n \to \mathbb{Q}^{n-j}$ the orthogonal projection along E (on the orthogonal complement $E^\perp$ of E). Note that since we consider vectors as column matrices of their coordinates on the canonical basis, computing the projection $\pi(x)$ of a vector x boils down to annihilating the coordinates $i_1, \ldots, i_j$ of x.

Theorem 19. *Assume that there are no constant positive linear combinations of the (L_i) and that $K' \cap \mathbb{Z}^n$ contains at least one point. Then if $\pi(K) \subseteq \pi(K')$, $K \cap K' \cap \mathbb{Z}^n$ contains infinitely many points.*

Proof. Let us first calculate the Smith normal form of the matrix A'. This gives U, D, and V, matrices over $\mathbb{Z}$ such that $UA'V = D$, where U and V are square matrices invertible over $\mathbb{Z}$, and D is diagonal (but not necessarily square).
Since U and V are invertible, we have:

$$\begin{aligned} \{x \in \mathbb{Q}^n \mid A'x = 0\} &= \{x \in \mathbb{Q}^n \mid UA'V(V^{-1}x) = 0\} \\ &= \{Vx \in \mathbb{Q}^n \mid Dx = 0\} \end{aligned}$$

Similarly, and since V is invertible over $\mathbb{Z}$, we have:

$$\begin{aligned} \{x \in \mathbb{Z}^n \mid A'x = 0\} &= \{Vx \mid x \in \mathbb{Z}^n \wedge Dx = 0\} \\ &= \{Vx \mid x \in \ker D \cap \mathbb{Z}^n\} \end{aligned}$$

By hypothesis, there exists x_0 a point of $K' \cap \mathbb{Z}^n$. For any point x of K', we have $A'(x - x_0) = (A'x + b') - (A'x_0 + b') = 0$. Therefore,

$$\begin{aligned} K' &= \{x_0 + Vx \mid x \in \ker D\} \\ K' \cap \mathbb{Z}^n &= \{x_0 + Vx \mid x \in \ker D \cap \mathbb{Z}^n\} \end{aligned}$$

Let $R = \max(1, \max_i(\sum_j |v_{i,j}|)) = \max(1, \max_i \|v_i\|_1)$ and N an arbitrary large integer. By Theorem 16, there exists a ball $B = B_\infty(w, R(N+1))$, of diameter $2R(N+1)$, contained in K. The projection $\pi(B)$ contains at least N^{n-j} points that are at least at distance $2R$ from each other and at least at distance R from its border. Each of these points has at least one antecedent by π in K' since $\pi(B) \subseteq \pi(K) \subseteq \pi(K')$. We call $\mathcal{T} \subset K'$ this set of points. The distance between any two points of $\mathcal{T}$ is at least $2R$ since the coordinates of the vectors in $\mathcal{T}$ on $E^\perp$ coincide with the ones of their respective projections.

For any point $y = x_0 + Vx \in K'$, there is a point of $K' \cap \mathbb{Z}^n$ at most at distance R. Indeed, since D is diagonal and $x \in \ker D$, any vector having the same non-zero coordinates as x remains in $\ker D$. Hence truncating the coordinates of x gives a vector $\lfloor x \rceil \in \ker D \cap \mathbb{Z}^n$. By definition of R, the distance between y and $x_0 + V\lfloor x \rceil$ is at most R.

Hence for each $t \in \mathcal{T}$, there exists $u \in K' \cap \mathbb{Z}^n$ such that the distance between t and u is at most R. But the distance between $\pi(t)$ and $\pi(u)$ is also at most R since

the projection only annihilates some coordinates. Therefore we obtain a family $\mathcal{U}$ of N^{n-j} distinct points of $K' \cap \mathbb{Z}^n$ that have a projection inside $\pi(B)$. For each $u \in \mathcal{U}$, $u = b + e$ where $b = \pi(u) \in \pi(B)$ and $e \in E$. Now since $\pi(B) \subset \pi(K)$ there exists $e' \in E$ such that $b + e' \in B \subset K$. Hence $u = b + e' + (e - e')$ belongs to K since K is invariant by the translation of direction $e' - e \in E$. Finally the N^{n-j} points of $\mathcal{U}$ are in $K \cap K' \cap \mathbb{Z}^n$, which concludes the proof when $j < n$.

In the degenerate case where $j = n$, K is either empty or the whole space. K cannot be empty since by Theorem 8 this would imply the existence of constant positive linear combination of the (L_i), hence a contradiction. Now K cannot be the whole space if there is no zero combination of the (L_i) which again contradicts the present hypothesis. □

3.4.3 Rational substitutions with Gaussian elimination

Let us now investigate the use of rational substitutions obtained with a simple Gaussian elimination algorithm on rational numbers at line 16. In this case, we do not introduce any slack variables but the coefficients involved in the substitutions are rationals, possibly non integers. The functions $check$ are implemented as follows:

- The function $check_1$ has now to test whether the set of equalities in Eq admits some solutions in $\mathbb{Z}$ (contrarily to the first scenario) and to return them. Indeed, we can have rational solutions for the relaxation problem we obtain when using Gaussian elimination but no integer solutions. In this case, there can be either zero integer solution, or one, or an infinite number of them.

- The function $check_\infty$ can take benefit of Theorem 19. The hypotheses of the theorem are actually verified thanks to the Gaussian elimination; and the vector space E of Theorem 19 is generated by the vectors of the canonical basis associated with the variables already substituted. Since these variables have been eliminated from C, the set of solutions of C is obviously invariant by translation along these coordinates. By construction of Eq the hypothesis of inclusion of the respective projections also holds. Therefore if the system of equalities Eq admits at least one integer solution, then there are an infinite number of integer solutions for the problem considered in the current branch. Otherwise there is no solution for this branch.

3.5 Examples

To get a flavor of our procedure, we illustrate its use through simple examples. We classify them into four categories regarding their satisfiability status and the way they are proved by the procedure. In the following, we consider:

1. a satisfiable example not requiring case-split analysis (example 20),
2. an unsatisfiable example not requiring case-split analysis (example 21),
3. a satisfiable example requiring case-split analysis (example 22),
4. an unsatisfiable example requiring case-split analysis (example 23).

Example 20. *Let us first consider the following inequalities where x and y are two integer variables:*

$$\begin{aligned} \mathit{green}: \quad & \underbrace{-761x + 366y - 445}_{L_1} \leq 0 \quad \wedge \\ \mathit{red}: \quad & \underbrace{887x + 563y + 427}_{L_2} \leq 0 \quad \wedge \\ \mathit{blue}: \quad & \underbrace{-317x + 111y - 291}_{L_3} \leq 0 \end{aligned}$$

They define the figure below. The gray part represents their rational solutions set. The main steps to prove their satisfiability are:

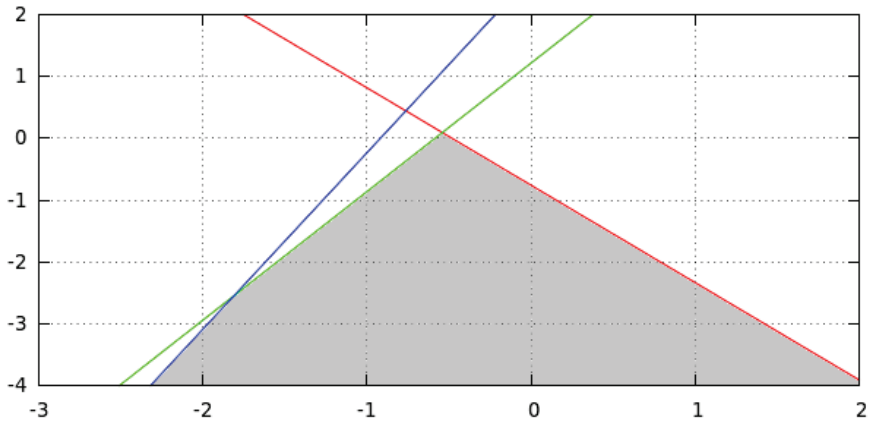

Figure 3.2: Satisfiable example without case-split analysis

- *The procedure calls the oracle with L_1, L_2 and L_3 at line 8. Following the method described in Section 3.2.2, the oracle attempts to find values for the rationals λ_1, λ_2 and λ_3 that are solution of the optimization problem given below in order to compute a particular constant positive linear combination of the affine forms L_1, L_2 and L_3:*

$$\begin{array}{lll} \mathit{maximize} & & -445\,\lambda_1 + 427\,\lambda_2 - 291\,\lambda_3 \\ \mathit{subject\ to} & (1) & -761\,\lambda_1 + 887\,\lambda_2 - 317\,\lambda_3 = 0 \quad \wedge \\ & (2) & 366\,\lambda_1 + 563\,\lambda_2 + 111\,\lambda_3 = 0 \quad \wedge \\ & (3) & \lambda_1 + \lambda_2 + \lambda_3 > 0 \quad \wedge \\ & (4) & \bigwedge_{i=1}^{3} \lambda_i \geq 0 \end{array}$$

- *The equality (2) necessarily implies that $\lambda_1 = \lambda_2 = \lambda_3 = 0$ modulo the non-negativeness constraints in (4). But, this solution is forbidden by (3). The oracle consequently answers that there is no constant positive linear combination of L_1, L_2 and L_3 and the initial problem is satisfiable.*

Let us use the method described in 3.2.3 to exhibit a model for this example:

- *We first define the vector $\gamma \in \mathbb{Q}^3$ as follows: $\gamma_1 = \frac{1127}{2}$, $\gamma_2 = 725$, $\gamma_3 = 214$,*
- *We look for a vector $w \in \mathbb{Q}^2$ that is solution of $Aw + b + \gamma \leq 0$, i.e. such that:*

$$\begin{array}{rlll} \mathit{green}^* : & -761w_1 + 366w_2 - 445 + \frac{1127}{2} \leq 0 & \wedge \\ \mathit{red}^* : & 887w_1 + 563w_2 + 427 + 725 \leq 0 & \wedge \\ \mathit{blue}^* : & -317w_1 + 111w_2 - 291 + 214 \leq 0 \end{array}$$

- *We use a rational simplex to get a valuation for w. In our case, we obtain the solution $w_1 = 0$, $w_2 = -\frac{1152}{563}$,*
- *Finally, the integer point $\widehat{w}$ is defined by $\widehat{w}_1 = 0$, $\widehat{w}_2 = -2$. The figure below shows the constraints defining K and K', and the ball $B_\infty(\frac{1}{2}, w)$ that is indeed a square in a 2D Euclidian orthonormal space.*

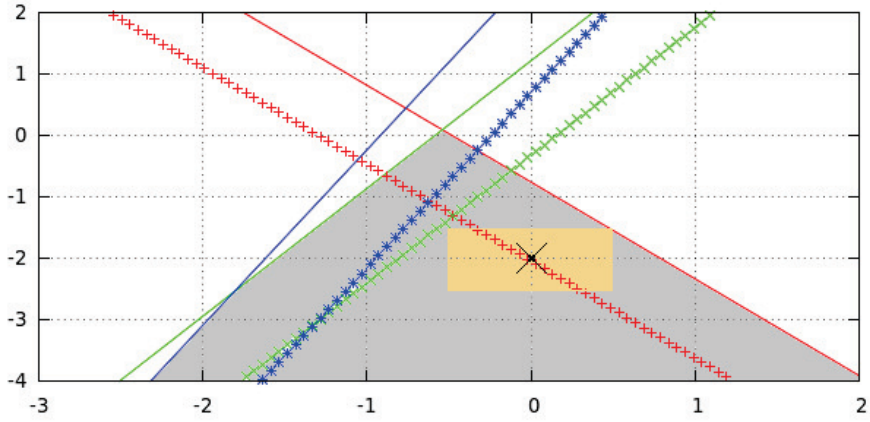

Figure 3.3: Satisfiable example without case-split analysis: model calculation

Example 21. *In this example, we consider the following formula:*

$$
\begin{aligned}
\mathit{green}: \quad & \underbrace{-361x + 366y + 445}_{L_1} \leq 0 \quad \wedge \\
\mathit{red}: \quad & \underbrace{-887x - 563y - 427}_{L_2} \leq 0 \quad \wedge \\
\mathit{blue}: \quad & \underbrace{317x - 111y + 291}_{L_3} \leq 0
\end{aligned}
$$

There is no gray part in the figure below because the rational solutions set of these constraints is empty. The main steps to prove their unsatisfiability are:

- *The procedure calls the oracle with L_1, L_2 and L_3 at line 8. Then, the oracle attempts to compute a particular constant positive linear combination of L_1, L_2 and L_3. For that, it tries to find values for the rationals λ_1, λ_2 and λ_3 that are solution of the*

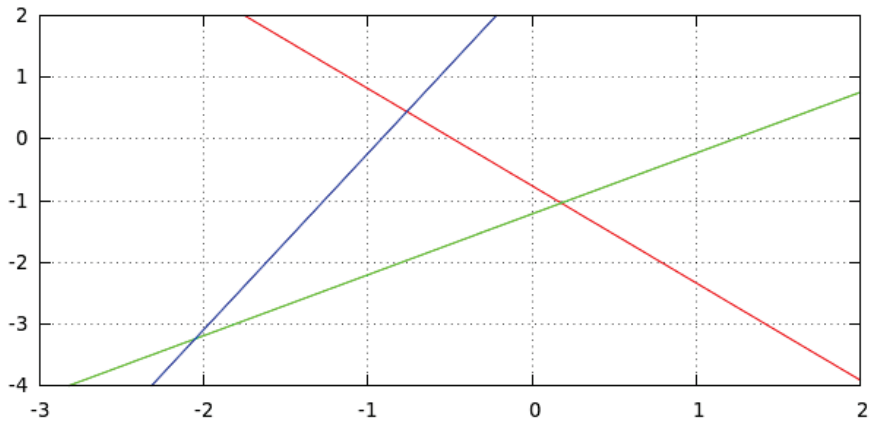

Figure 3.4: Unsatisfiable example without case-split analysis

following optimization problem:

$$
\begin{array}{lll}
\textit{maximize} & & 445\,\lambda_1 \;-\; 427\,\lambda_2 \;+\; 291\,\lambda_3 \\
\textit{subject to} & (1) \quad -361\,\lambda_1 \;-\; 887\,\lambda_2 \;+\; 317\,\lambda_3 \;=\; 0 & \wedge \\
& (2) \quad 366\,\lambda_1 \;-\; 563\,\lambda_2 \;-\; 111\,\lambda_3 \;=\; 0 & \wedge \\
& (3) \quad \lambda_1 \;+\; \lambda_2 \;+\; \lambda_3 \;>\; 0 & \wedge \\
& (4) \quad \textstyle\bigwedge_{i=1}^{3} \lambda_i \;\geq\; 0
\end{array}
$$

- *A slack variable s is introduced and the constraint (3) is replaced by:*

$$
\begin{array}{lll}
(4) & s \;=\; \lambda_1 \;+\; \lambda_2 \;+\; \lambda_3 \;-\; \varepsilon & \wedge \\
(5) & s \;\geq\; 0
\end{array}
$$

  where ε is a positive rational parameter.

- *The equalities are then solved in the rationals modulo the non-negativeness constraints (4) and (5). We obtain for instance the following solved form:*

$$
\begin{cases}
\lambda_1 \;\mapsto\; \frac{276928}{527885}\,\lambda_3 \\
\lambda_2 \;\mapsto\; \frac{75951}{527885}\,\lambda_3 \\
s \;\mapsto\; \frac{880764}{527885}\,\lambda_3 \;-\; \varepsilon
\end{cases}
$$

- *Now, we initialize the simplex algorithm with this matrix and try to maximize the objective function which is normalized to* $\frac{244416418}{527885}\ \lambda_3$,

- *The simplex detects that the problem is unbounded and returns an arbitrary positive constant. The procedure consequently deduces that the initial conjunction is unsatisfiable.*

Example 22. *Now, let us consider the following conjunction defining the gray triangle shown in the figure below:*

$$\begin{aligned}
green: \quad & \underbrace{361x - 366y - 445}_{L_1} \leq 0 \quad \wedge \\
red: \quad & \underbrace{887x + 563y + 427}_{L_2} \leq 0 \quad \wedge \\
blue: \quad & \underbrace{-317x + 111y - 291}_{L_3} \leq 0
\end{aligned}$$

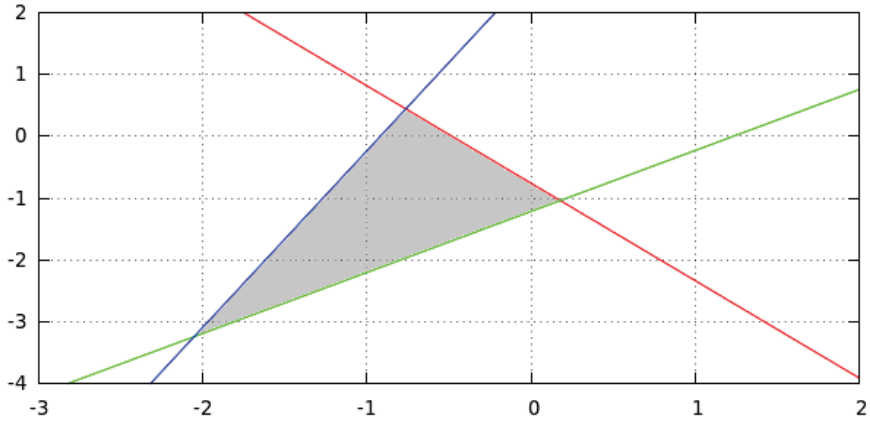

Figure 3.5: Satisfiable example with case-split analysis

The main steps to prove the satisfiability of this conjunction are:

- *At line 8, the oracle tries to compute a particular constant positive linear combina-*

tion of L_1, L_2 and L_3 by solving the following optimization problem:

$$\begin{array}{lll} \mathit{maximize} & & -445\,\lambda_1 + 427\,\lambda_2 - 291\,\lambda_3 \\ \mathit{subject\ to} & (1) & 361\,\lambda_1 + 887\,\lambda_2 - 317\,\lambda_3 = 0 \quad \wedge \\ & (2) & -366\,\lambda_1 + 563\,\lambda_2 + 111\,\lambda_3 = 0 \quad \wedge \\ & (3) & \lambda_1 + \lambda_2 + \lambda_3 > 0 \quad \wedge \\ & (4) & \bigwedge_{i=1}^{3} \lambda_i \geq 0 \end{array}$$

- *The maximum of this problem is equal to $-\frac{6432011}{23178}\,\varepsilon$. The corresponding valuation of λ is:*

$$\lambda_1 = \tfrac{69232}{220191}\,\varepsilon, \quad \lambda_2 = \tfrac{25317}{293588}\,\varepsilon, \quad \lambda_3 = \tfrac{527885}{880764}\,\varepsilon$$

- *These information allow us to deduce lower bounds for the initial affine forms:*

$$\begin{array}{rcccccc} -882 & = & \lceil -\frac{6432011}{23178} \cdot \frac{220191}{69232} \rceil & \leq & L_1 & \leq & 0 \\ -3218 & = & \lceil -\frac{6432011}{23178} \cdot \frac{293588}{25317} \rceil & \leq & L_2 & \leq & 0 \\ -463 & = & \lceil -\frac{6432011}{23178} \cdot \frac{880764}{527885} \rceil & \leq & L_3 & \leq & 0 \end{array}$$

- *Assume that the procedure performs a case-split analysis on L_3. The values of L_3 in $[\![-463, -403]\!]$ lead to unsatisfiability. But, the value $L_3 = -402$ satisfies the conjunction. Indeed, solving this equality yields the substitution $\sigma = \{x \mapsto -111k,\ y \mapsto -317k - 1\}$ which, when applied on L_1 and L_2, yields:*

$$\begin{array}{rcccccc} -882 & \leq & \underbrace{+75951k - 79}_{L_1\sigma} & \leq & 0 & \wedge \\ -3218 & \leq & \underbrace{-276928k - 136}_{L_2\sigma} & \leq & 0 & \end{array}$$

Now, the only value for k that satisfies these constraints is $k = 0$. We thus obtain the model $x = 0,\ y = -1$ for this example.

Example 23. *Finally, consider the following formula:*

$$
\begin{aligned}
green: \quad & \underbrace{761x - 366y + 445}_{L_1} \leq 0 \quad \wedge \\
red: \quad & \underbrace{887x + 563y + 427}_{L_2} \leq 0 \quad \wedge \\
blue: \quad & \underbrace{-317x + 111y - 291}_{L_3} \leq 0
\end{aligned}
$$

Geometrically, these inequalities define the figure below. They are satisfiable in the rationals but not in the integers. The gray triangle represents their solutions set.

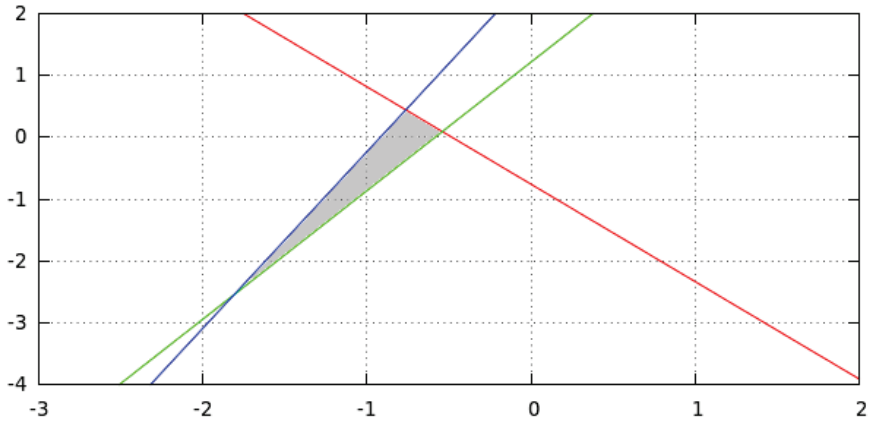

Figure 3.6: Unsatisfiable example with case-split analysis

This formula is proved by the same reasoning used for example 22 except that no satisfying integer assignment will be found.

3.6 Implementation in CTRL-ERGO

In general, formulas that are sent to SMT-sovlers are not just conjunctions of literals. They mix boolean connectives and high-level constructs in a non-trivial way. For instance, more than 50% of formulas in the QF-LIA benchmark of SMT-LIB [18]. (*e.g.* NEC families) intensively use high level constructs such as LET-IN and IF-THEN-ELSE. The preprocessor and the SAT solver of ALT-ERGO are not tuned for this kind of formulas.

To measure the effectiveness of our simplex-based procedure, we implemented it in a small prototype, called CTRL-ERGO. This SMT solver is based on an OCAML imperative implementation of MINISAT. Furthermore, it integrates a home-made simplification step for IF-THEN-ELSE and LET-IN constructs. In this section, we provide some details about this implementation.

3.6.1 Preprocessing

In this section, we describe the technique used in CTRL-ERGO to simplify formulas containing LET-IN and IF-THEN-ELSE constructs. Note that, some SMT solvers incorporate preprocessing techniques for such constructions [77].

Consider the example given in Figure 3.7, written in the syntax of the SMT-LIB-2 language [18].

```
1 (let ((v1 (ite (= a 1) (ite (>= (+ a a) 3) 1 2) 3)))                ;; in
2   (let ((v2 (ite (>= 1 0) (ite (= (+ v1 1) 2) a v1) 6)))           ;; in
3     (let ((v3 (ite (not (= a 1)) (ite (>= (- v1 v2) 0) 2 4) v2))) ;; in
4       (= v1 (ite (= v1 2) v1 (ite (= v3 1) a v1))))))
```

Figure 3.7: A simple overview of the formulas in NEC families

This formula cannot be expressed in the language of many-sorted first-order logic described in Chapter 2, because the imbrication of formulas in terms is not allowed. However, it can be expressed using the grammar of Figure 3.8, where τ produces terms, π produces non negated atoms, α produces literals, φ (*resp.* ψ) produces formulas without (*resp.* with) LET-IN and ν (*resp.* ζ) produces integer variables (*resp.* integer numerals).

$$
\begin{array}{lcllll}
\tau & ::= & \nu & \mid \zeta + \sum(\zeta \cdot \nu) & \mid \text{ITE}(\alpha, \tau, \tau) \\
\pi & ::= & \text{TRUE} & \mid \nu & \mid \tau \bowtie 0 \\
\alpha & ::= & \pi & \mid \text{NOT}\ (\pi) & \\
\varphi & ::= & \alpha & \mid \text{AND}\ (\varphi^*) & \mid \text{OR}\ (\varphi^*) \\
\psi & ::= & \varphi & \mid \text{LET-IN}(\nu, \tau, \psi) & \mid \text{LET-IN}(\nu, \varphi, \psi) \\
\nu & ::= & \text{STRING} & & \\
\zeta & ::= & \mathbb{Z} & & \\
\bowtie & ::= & \approx & \mid \geq &
\end{array}
$$

Figure 3.8: The grammar of terms, propositional variables, atoms and formulas

The elimination of LET-IN and IF-THEN-ELSE occurrences in the example above, before any other simplification[1], yields the equivalent formula shown in Figure 3.9.

```
1  (and
2    (or
3      (and (not (= v1 1)) (= (+  v2 (- v1)) 0))
4      (and (= v1 1) (= (+  v2 (- a)) 0))
5    )
6    (or
7      (and (not (= a 1)) (not (>= (+  v2 (- v1)) 1)) (= v3 2))
8      (and (not (= a 1)) (>= (+  v2 (- v1)) 1) (= v3 4))
9      (and (= a 1) (= (+  v3 (- v2)) 0))
10   )
11   (or
12     (and (not (= a 1)) (= v1 3))
13     (and (= a 1) (= v1 2))
14   )
15   (or
16     (= v1 2)
17     (not (= v3 1))
18     (and (= (+  v1 (- a)) 0) (not (= a 2)))
19   )
20 )
```

Figure 3.9: The result of a naive elimination of LET-IN and IF-THEN-ELSE

As illustrated by this example, a naive elimination approach blows up the size of resulting formulas. We describe below a preprocessing step that simplifies input formulas in order to limit the magnitude of this explosion.

Canonization

We put terms, predicates, literals and formulas in a canonical form. This allows us to easily detect duplicates and complementary literals (*resp.* formulas). For instance, the formula in Figure 3.7 is not in canonical form:

- at line 1, the predicate $a + a \geq 3$ can be simplified to $a \geq 2$ modulo LIA
- at line 2, the predicate $v_1 + 1 \approx 2$ can be simplified to $v_1 \approx 1$ modulo LIA
- at line 2, $1 \geq 0$ is $true$ modulo LIA. Thus, the ELSE branch can be pruned
- at line 3, the term of the form `(ite (not p) a b)` simplifies to `(ite p b a)`

[1]Replacement of LET-IN with equalities and IF-THEN-ELSE with combinations of AND and OR

All these simplifications yield the following "canonized" formula:

```
1 (let ((v1 (ite (= a 1) (ite (>= a 2) 1 2) 3)))                    ;; in
2   (let ((v2 (ite (= v1 1) a v1)))                                 ;; in
3     (let ((v3 (ite (= a 1) v2 (ite (>= (- v1 v2) 0) 2 4))))       ;; in
4       (= v1 (ite (= v1 2) v1 (ite (= v3 1) a v1))))))
```

Figure 3.10: The canonization of the formula in Figure 3.7

Contextual simplification

Our simplification technique interleaves normalization with contextual learning (theory learning and forward/backward propagation). A basic, incomplete but fast, procedure for QF-LIA is used to transform assumed equalities into a term rewriting system, and to perform a cheap interval/domain calculus on assumed inequalities and disequalities. Contextual learning allows to deduce more precise domains for intermediate variables. The normalization of terms, predicates, atoms and formulas is performed modulo the current context provided by the state of the procedure for QF-LIA. Using these techniques, the formula in Figure 3.10 is reduced to $true$ as follows:

1. Using a very basic arithmetic reasoning, we realize that v_1 cannot be equal to 1 because $a \approx 1 \wedge a \geq 2$ is inconsistent. Consequently, line 1 becomes `let ((v1 (ite (= a 1) 2 3)))`. Moreover, we can deduce that $v_1 \in \{ 2, 3 \}$.
2. Since $v_1 \not\approx 1$, the LET-IN construct at line 2 simply rewrites to $v_2 \approx v_1$.
3. Now, we first use the equality $v_1 \approx v_2$ to rewrite the third LET-IN as follows: `(let ((v3 (ite (= a 1) v1 2)))`. Then, when $a \approx 1$ is $true$, v_1 is equal to 2. Consequently v_3 is equal to 2 whatever the value of a.
4. The formula at line 4 first reduces to `(= v1 (ite (= v1 2) v1 v1))` because $v_3 \not\approx 1$. Then, it simplifies to $true$ whatever the value of v_1.
5. Finally, we throw away the declarations of the intermediate variables v_1, v_2 and v_3 because they are not used in the formula $true$.

3.6.2 The SAT solver

We now describe our OCAML implementation of MINISAT using inference rules. These rules are inspired by those present in the literature [48, 82, 98]. But, they are sufficiently low level to reflect the actual implementation.

Let Δ_0 be a set of clauses, where each clause is represented by a set of literals. The SAT solver attempts at constructing a model Γ for the CNF formula represented by Δ_0. Recall that a set with an empty clause in unsatisfiable and an empty set of clauses in satisfiable.

The model Γ is represented by a list of literals of the form $l_C^n :: l_D^m :: \cdots ::$ `Empty`. It is thought of as a conjunction of literals. Every literal in Γ is implicitly assigned to $true$[2]. It is annotated with a natural number called its decision/implication level and a possibly empty clause explaining why the literal is assigned to $true$.

We use the following notations in the presentation below: given a literal l and a clause C, $\texttt{undef}(l, \Gamma)$ means that neither l nor $\neg l$ are in Γ. We denote by $\texttt{lvl}(l, \Gamma)$ the current level of l or $\neg l$ in Γ. It is equal to -1 if $\texttt{undef}(l, \Gamma)$. The level of C, denoted $\texttt{lvl}(C, \Gamma)$, is the maximal level of its literals. We denote by $\texttt{lvl}(\Gamma)$, the level of the left-most literal in Γ. The level of `Empty` is equal to 0. The notation $\Gamma|_n$ designates the sub-list of Γ containing literals of level at most n. The notation $l \in \Delta$ is an abbreviation for: there is a clause $C \in \Delta$ such that $l \in C$ or $\neg l \in C$. We use functions of the form `xxx-heuristic` to indicate that a side-condition depends on an heuristic. The SAT solver works on configurations of the form $\Gamma \vdash \Delta$ in *search mode* and of the form $\Gamma \nvdash^n \Delta : C$ in *conflict mode*, where n is the conflict level.

Search mode. The inference rules in search mode are given in Figure 3.11. The first (*resp.* second) rule detects when a model (*resp.* an inconsistency) is found. The third rule adds new literals to Γ thanks to function `next-lit` (see below). The rule 4 removes clauses from Δ when they are satisfied by $\Gamma|_0$. The rule 5 removes learned clauses from Δ when some conditions are satisfied. The rule 6 restarts the search from an empty model while keeping learned clauses. Finally, the algorithm enters in conflict mode when a clause in Δ is falsified by Γ (rule 7).

The behavior of the function `next-lit` is described in Figure 3.12. The inference rules only apply on a literal l such that $l \in \Delta$ and $\texttt{undef}(l, \Gamma)$. We notice that implied literals (rule 8) are annotated with Γ's current level, whereas the level of decided literals (rule 9) is incremented by 1. In practice, the choice of the next literal to decide is guided by the dynamic variables activity (VSID) heuristic. Note that, the list Γ is always sorted in decreasing order *w.r.t.* the level of literals.

The strategy used in search mode is sketched in Figure 3.13. The function `search-mode` repeatedly propagates assigned facts, unless a restart is scheduled (line 2) or a conflict is encountered (line 3). In the latter case, the function raises the exception `Unsat` if the conflict is derived without any assumption or calls

[2]Hence, the negation of this literal is assigned to $false$.

	name	premise	conclusion	side conditions
1.	**sat**	$\Gamma \vdash \Delta$	Γ	$\Gamma \models \Delta$
2.	**unsat**	$\Gamma \not\vdash \Delta : \bot$	$\bot$	
3.	**assume**	$\Gamma \vdash \Delta$	$\Gamma' \vdash \Delta$	$\texttt{next-lit}(\Gamma \vdash \Delta) = \Gamma' \neq \Gamma$
4.	**simplify**	$\Gamma \vdash \Delta, C$	$\Gamma \vdash \Delta$	$\Gamma\vert_0 \models C$
5.	**forget**	$\Gamma \vdash \Delta, C$	$\Gamma \vdash \Delta$	$C \notin \Delta_0$ and $\texttt{rdb-heuristic}(C)$
6.	**restart**	$\Gamma \vdash \Delta$	$\texttt{Empty} \vdash \Delta$	$\texttt{restart-heuristic}()$
7.	**conflict**	$\Gamma \vdash \Delta, C$	$\Gamma \not\vdash \Delta, C : C$	$\Gamma \models \neg C$

Figure 3.11: Computation rules in search mode

	name	premise	conclusion	side conditions
8.	**propagate**	$\Gamma \vdash \Delta \cup \{D \vee l\}$	$l_D^{\vert\Gamma\vert} :: \Gamma$	$\Gamma \models \neg D$
9.	**decide**	$\Gamma \vdash \Delta$	$l_\emptyset^{\vert\Gamma\vert+1} :: \Gamma$	$l = \texttt{vsid-heuristic}()$

Figure 3.12: Assigning the next literal

```
1  let search_mode () =
2    while not !restart do
3      match propagate () with                          (* 8. propagate *)
4        | Some c ->
5          if decision_level() = 0 then raise Unsat;    (* 2. unsat     *)
6          conflict_mode c                              (* 7. conflict  *)
7
8        | None   ->
9          if all_variables_assigned () then raise Sat; (* 1. sat       *)
10         if reduce_db_heuristic () then reduce_db (); (* 5. forget    *)
11         if decision_level() = 0 then simplify ();    (* 4. simplify  *)
12         restart := restart_heuristic ();
13         if !restart then restart_search ()           (* 6. restart   *)
14         else decide ()                               (* 9. decide    *)
15   done
16
17 let solve () =
18   while true do
19     search_mode ();
20   done
```

Figure 3.13: The strategy used is search mode

`conflict-mode`, otherwise. When all the propagations of the current decision level are done (line 8), the function raises the exception `Sat` if all the literals are assigned. Otherwise, it heuristically removes some learned clauses from its database, and removes the clauses that are valid at level 0. Then, it either schedule a restart or assigns a new decision literal. The function `solve` consists of an infinite loop that repeatedly re-calls `search-mode` after each restart, unless an exception is raised.

Conflict mode. The inference rules in conflict mode are shown in Figure 3.14. The rule 10 applies a resolution step between the clause attached to the left-most literal in Γ and the current conflicting clause. This rule is only applicable if there exists another literal in C of level n. If not, or if resolution does not apply, the rule 11 is used to pop the top of Γ. The two other rules are used when the level of Γ is lower than n. The rule 12 also pops literals from the top of Γ. It unassigns implied and decided literals until the right backjump level is found. The rule 13 is then applied to learn the clause C. This clause is of level m and contains exactly one unassigned literal. The SAT solver then goes back to search mode. Note that, there is one possible strategy for these rules, because the side conditions are mutually exclusive. Moreover the activity of literals and learned clauses that are used in conflict resolution is augmented.

	name	premise	conclusion	side conditions
10.	**resolve**	$l^n_D :: \Gamma \not\vdash^n \Delta : \neg l \vee C$	$\Gamma \not\vdash^n \Delta : C \vee D$	$\texttt{lvl}(C, \Gamma) = n$
11.	**skip**	$l^n_D :: \Gamma \not\vdash^n \Delta : C$	$\Gamma \not\vdash^n \Delta : C$	$\neg l \notin C$ or $\texttt{lvl}(C,\ \Gamma) < n$
12.	**undo**	$l^m_D :: \Gamma \not\vdash^n \Delta : C$	$\Gamma \not\vdash^n \Delta : C$	$m < n$ and $\texttt{lvl}(C,\ l^m_D :: \Gamma) \neq m$
13.	**learn& search**	$\Gamma \not\vdash^n \Delta : C$	$\Gamma \vdash \Delta \cup \{C\}$	$m < n$ and $\texttt{lvl}(C,\ \Gamma) = m$

Figure 3.14: Computation rules in conflict mode

The rules given in Figure 3.15 integrate theory reasoning in the SAT solver. The notation $\mu \subseteq \Gamma$ is thought of as set inclusion. The rule 7b detects theory conflicts: when a subset μ of Γ is inconsistent modulo theory, the SAT solver enters in conflict mode with the conflicting clause $\neg\mu$. Moreover, the function `next-lit` is extended with the rule 8b to enable the propagation of implied literals at the theory level.

	name	premise	conclusion	side conditions
7b.	**T-conflict**	$\Gamma \vdash \Delta$	$\Gamma\vert_n \not\vdash^n \Delta : \neg\mu$	$\exists \mu \subseteq \Gamma.\ \mu \models_{\mathrm{T}} \bot$ and $n = \mathtt{lvl}(\neg\mu, \Gamma)$
8b.	**T-propagate**	$\Gamma \vdash \Delta$	$l^{\vert\Gamma\vert}_{\neg\mu} :: \Gamma$	$\exists \mu \subseteq \Gamma.\ \mu \models_{\mathrm{T}} l$

Figure 3.15: Integration of theory reasoning

3.6.3 The decision procedure for QF-LIA

We now describe the implementation of our decision procedure for QF-LIA. The simplified interface used by the SAT solver to interact with the procedure is shown in Figure 3.16. The function `assume` is used to add assigned literals at the SAT level to theory's environment. This function raises the exception `Inconsistent` when a theory conflict is detected. The function `learn` performs theory propagation. The function `case_split` handles case-split analysis. Finally, the function `empty` is used to construct empty theory environments.

```
1 type env
2 exception Inconsistent of atom list
3
4 val assume : env -> atom -> env
5 val learn  : env -> atom
6 val case_split : env -> env
7 val empty : unit -> env
```

Figure 3.16: The simplified interface of the decision procedure

Actually, our implementation is much more complicated than the interface above. It is made of the modules shown in Figure 3.17. The data structures used in this part of the solver are persistent, except for the `Simplex`. Backtracking is therefore immediate for persistent structures, and the `Simplex`'s environment is reconstructed from scratch, since it is currently not incremental nor backtrackable.

The module `Theory` provides the interface shown in Figure 3.16. In addition, it implements *dynamic clustering* as follows: `Theory.env` is a list of environments (clusters) provided by `Lia`. These individual "clusters" do not share variables. When a literal that involves variables appearing in different clusters is assumed, these clusters are merged before the literal is treated.

`Lia`'s environment is a pair, where the first component is provided by `Lia_eqs` and the second one by `Lia_ineqs`. Equalities are assumed in the first component. Inequalities and disequalities are assumed in the second one. `Lia` implements

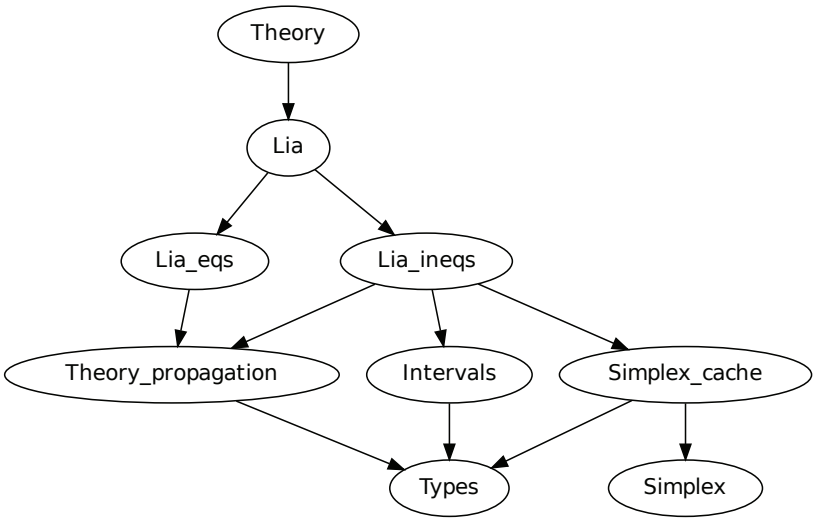

Figure 3.17: The architecture of the decision procedure's modules

case-split analysis as a recursive function with non-chronological backtracking and uses a heuristic that privileges affine forms with smaller intervals. This technique is similar to that described in Chapter 5.

The module `Lia_eqs` handles equalities using a rewriting system, encoded by a persistent dictionary from variables to arithmetic terms. It relies on a solver that builds substitutions with integer slack variables [80]. We have also a version that uses a rational solver as described in Section 3.4. However, it is less efficient.

Inequalities and disequalities are handled by `Lia_ineqs`. They are added to a dictionary associating affine forms to intervals of integers. The affine forms are maintained in normal form *w.r.t.* the rewriting system provided by `Lia_eqs`. The intervals of integers are handled by the module `Intervals`. The `Simplex` module is used to infer new bounds as described in Section 3.2.2. Note that, our actual implementation of the oracle is not incremental. To alleviate, we use memoization techniques (module `Simplex_cache`) to reuse previously computed results at the expense of a larger memory footprint.

`Theory_propagation` implements an incomplete, but fast, theory propagation mechanism. The module `Types` provides data structures for representing terms, literals, CNF formulas, etc.

Note that, in earlier implementations, we used linear optimization problems similar to that presented in Section 3.2.2 to deduce refined bounds for the uninterpreted constants appearing in the affine forms. However, we have deactivated this extension because it does not scale on large problems.

3.7 Experimental results

In this section, we benchmark CTRL-ERGO and compare its performances with state-of-the-art SMT solvers including MATHSAT5 [66] (v5.1.3), Z3 [42] (v3.2) and YICES2 [41] (v2.0-prototype). We could not include the MISTRAL solver that implements the procedure of [53], because it was not possible to obtain it.

We have used the QF-LIA benchmark for our experiments. In particular, we focused on a test suite containing 1070 composed of the families listed below. We have selected these families because they are known to be well-suited for stressing the integer-reasoning part of SMT solvers.

- CAV-2009: randomly-generated instances used in [53]. Most of them are satisfiable. They are reported very hard for modern SMT solvers,
- SLACKS: reformulation of CAV-2009 instances used in [73] that introduces slack variables to bound all variables,
- CUT-LEMMAS: crafted instances encoding the validity of cutting planes in $\mathbb{Z}$,
- PRIME-CONE: crafted instances used in [73] that encode a tight n-dimensional cone around the point whose coordinates are the first n prime numbers.
- PIDGEONS (*sic*): crafted instances encoding the pigeonhole principle. They are reported hard for any solver not using cutting planes [73],
- PB2010: industrial instances coming from the PB competition (2010),
- MIPLIB2003: instances generated from some optimization problems in [2].

All measures were obtained on a 64-bit machine with a quad-core Intel Xeon processor at 3.2 GHz and 24 GB of memory. Provers were given a time limit of 600 seconds and a memory limit of 2 GB for each test. The results of our experiments are reported in Figure 3.18. The first two columns show the families we considered and the number of their instances. For each prover, we report both the number of solved instances within the required time for every family and the time needed for solving them (not counting timeouts). The last rows summarize the total number of solved instances and the accumulated time for each prover.

Although the first two families were reported very hard for modern SMT solvers, our approach only requires 320 seconds to solve almost all the instances. Thus,

[3] The time limit is 180 seconds for the tests of the complete benchmark.

SMT SOLVERS		CTRL-ERGO		MATHSAT5		MATHSAT5+CFP		YICES 2		Z3	
families	**#inst.**	**solved**	**time**	**solved**	**time**	**solved**	**time**	**solved**	**time**	**solved**	**time**
CAV-2009	591	590	**253**	588	4857	589	4544	386	11664	590	5195
SLACKS	233	233	**67**	166	3551	155	6545	142	6102	187	9897
CUT-LEMMAS	93	93	**216**	62	3424	59	2775	92	1892	67	3247
PRIME-CONE	37	37	**0.4**	37	1	37	2.2	37	2.3	37	14
PIDGEONS	19	19	2	19	0.16	19	0.16	19	**0.01**	19	0.28
PB2010	81	23	390	38	743	34	1540	25	8.3	64	1831
MIPLIB2003	16	2	34.7	12	432	12	501	11	145.4	12	**241**
total	1070	997	**963.1**	922	13008	905	15907	712	19814	976	20425
total QF-LIA[3]	5882	4410	68003	5597	**47635**	5524	50481	3220	71324	5597	54503

Figure 3.18: Experimental results. Underlined values are for tools that have proved the most instances. Bolded results are for tools that have proved both the most instances and the fastest.

it significantly outperforms the other solvers' approaches. This observation also applies for the third and the fourth families. From the results of the sixth and the seventh families, we notice that our technique does not perform well on large difference-logic-like problems compared to MATHSAT5 and Z3's. We think this is partly due to our naive implementation of the simplex algorithm which computes on dense matrices while sparse matrices would be better suited for these problems.

The last row of Table 3.18 shows the results for the whole QF-LIA benchmark. There are two reasons for the poor results. First, CTRL-ERGO has to be tuned for parts other than the LIA solver. Second, some families, *e.g.* BOFILL, contain large intervals that need splitting, and the decision procedure does not deal efficiently with them. This would possibly require a combination of our approach with other established techniques for integers to cut down the search space.

3.8 Related and future works

Related works

Designing efficient decision procedures for QF-LIA has been an active research topic in the SMT community over the last decade. An efficient integration of the simplex algorithm in the DPLL(T) framework has been proposed in [55]. This integration rests on a preprocessing step that enables fast backtracking and efficient theory propagation. The contribution of [53] is seen as a generalization of *branch-and-bound*. Using the notion of *the defining constraints* of a vertex, it derives additional inequalities that prune higher dimensional subspaces not containing integer solutions. In our setting, the simplex algorithm is used on auxiliary problems to refine the search space by bounds inference.

The approach described in [65] focuses on combining several existing techniques using heuristics and *layering* to take advantage of each of them. We believe that the ideas we described in this chapter can naturally be used to enhance this combination approach. Another different contribution described in [73] consists in extending the inference rules of the CDCL procedure with linear arithmetic reasoning. This tight integration naturally takes advantage of good CDCL properties, such as model search, dynamic variable reordering, propagation, conflicts explanation, and backjumping. Note that there has also been some works on this topic in the constraint solvers community.

Future works

As reflected by our contribution, the simplex we use works on an auxiliary problem that simulates a run of the Fourier-Motzkin method, but not on the original problem nor on its dual. Therefore, fast incrementality and efficient backtracking techniques developed for simplex-based approaches are not suitable for our setting. We plan to investigate this issue in the future for a better integration of our approach with DPLL(T). In fact, memoization techniques used in our actual implementation are not satisfactory.

We also envisage to extend our method with a better conflict resolution/learning technique, and a cleverer case-split analysis mechanism. In particular, we think that the extension of our framework with the ideas developed in [73] and in [79] for the rationals would greatly improve it.

We believe that a combination of our approach with "well tuned" state-of-the-art techniques, *à la* MATHSAT, would be very beneficial. Furthermore, the use of advanced data-structures and algorithms such as sparse matrices and the revised simplex method [110] would greatly enhance our implementation.

CHAPTER 4

Ground AC Completion Modulo a Shostak Theory

In this chapter, we investigate the integration of Shostak theories in ground AC completion. We present a new combination framework, called AC(X), that decides ground conjunctions of equalities in the union of the free theory of equality with user-defined AC symbols, uninterpreted symbols and an arbitrary signature-disjoint Shostak theory X. Our combination technique relies on *canonized rewriting*, a new rewriting relation reminiscent to normalized rewriting [90] that integrates canonization in standard rewriting.

In Section 4.1, we first recall some notions about term rewriting systems and rewriting modulo AC. Then, we recall the inference rules of ground AC completion and illustrate its use throughout an example.

In Section 4.2, we first consider the combination of individual canonizers to handle mixed terms. Then, we define the notion of canonized rewriting. After that, we tackle the problem of solving equalities on heterogeneous terms. Finally, we pose some reasonable ordering restrictions on the canonizers and the solvers to prove the termination of our combination framework.

In Section 4.3, we show how to extend, in a modular and non intrusive way, the ground AC completion procedure with Shostak theories. The main ideas of our integration are the substitution of standard rewriting with canonized rewriting and the replacement of the equalities orientation mechanism found in ground AC completion with the solver for X. We then illustrate the AC(X) combination framework on a simple example.

In Section 4.4, we establish the correctness and termination of AC(X). Note that, our modular integration allows us to partly reuse the proofs of the ground AC completion procedure.

In Section 4.5, we show that a simple preprocessing step *à la* Bachmair et al. [6] enables the use a partial multiset ordering instead of an AC-compatible reduction ordering.

In Section 4.6, we measure the performances of our framework and compare it with axiomatization/instantiation based approaches. We also show that the instantiation mechanism of our ALT-ERGO SMT solver have to be extended modulo AC in order to fully integrate AC(X) at its core.

In Section 4.7, we discuss the most important extensions we have investigated and the issues we have encountered.

In Section 4.8, we overview related works and summarize some further directions to investigate.

A substantial part of this chapter has been published in [31, 32, 33].

4.1 Preliminaries

In this section, we recall the standard notations and definitions of [5, 50] for term rewriting systems. Let $\Sigma = (\mathcal{F}, \mathcal{P}, \mathcal{S}, ar, \tau)$ be a signature, where $\mathcal{P}$ contains only equality predicate symbols. Let $\mathcal{X}$ be a set of variables disjoint from $\mathcal{F}$ and $\mathcal{P}$.

4.1.1 Term rewriting

In rewriting community, a Σ-identity refers to an equality over Σ where all the free variables are implicitly universally quantified. Let E be a set of Σ-identities. The equational theory of E, denoted $\approx_E$, is the set of all identities that can be obtained by reflexivity, symmetry, transitivity, congruence and instances of identities in E. More formally, $\approx_E$ is defined as follows:

$$\approx_E \quad = \quad \left\{ \quad (s,t) \in \mathcal{T}_\Sigma(\mathcal{X}) \times \mathcal{T}_\Sigma(\mathcal{X}) \qquad \text{where} \quad \models_{\mathcal{E}} \quad E \Rightarrow s \approx t \quad \right\}$$

Note that, the equational theory of the free theory of equality $\mathcal{E}$, defined by the empty set of equations, is simply denoted $\approx$ instead of $\approx_{\mathcal{E}}$.

Given two ground terms s and t over Σ, the word problem for E consists in determining whether the equality $s \approx t$ belongs to the set $\approx_E$. If so, we simply write $s \approx_E t$. We say that the word problem for E is ground when E contains only ground identities. An equational theory $\approx_E$ is *inconsistent* when $s \approx_E t$, for any terms s and t in $\mathcal{T}_\Sigma$.

A rewriting rule is a pair of terms usually denoted by $l \to r$. In addition, we usually require that l is not a variable and that $\mathcal{V}ars(r) \subseteq \mathcal{V}ars(l)$. A term rewriting system (TRS) is a set of rewriting rules. The definition below introduces the notion of term rewriting:

Definition 24. *We say that a term s rewrites to a term t at position p by the rule $l \to r$, denoted by $s \to^{p}_{l \to r} t$, if and only if there exists a substitution σ such that*

$$s|_p = l\sigma \quad \textit{and} \quad t = s[r\sigma]_p$$

Let R be a term rewriting system. We write $s \to_R t$ whenever there exists a rule $l \to r$ in R such that s rewrites to t by $l \to r$ at some position. A normal form of a term s w.r.t to R is a term t such that $s \to^*_R t$ and t cannot be rewritten by R. The term rewriting system R is *terminating* if it does not admit any infinite reduction. It is *confluent* if every term admits at most one normal form. It is *convergent* if it is both terminating and confluent. In this case, every term admits exactly a unique normal form.

A well-founded quasi-ordering [49] $\preceq$ on terms is a reduction quasi-ordering if, for any substitution σ, for any term l and position p

$$s \preceq t \qquad \text{implies} \qquad \begin{cases} 1. \quad l[s]_p \preceq l[t]_p & \text{and} \\ 2. \quad s\sigma \preceq t\sigma \end{cases}$$

These conditions mean that $\preceq$ is (1) closed under context and (2) closed under substitutions. A quasi-ordering $\preceq$ defines an equivalence relation $\simeq$ as $\preceq \cap \succeq$ and a partial ordering $\prec$ as $\preceq \cap \not\simeq$. It has been shown [5] that a term rewriting system R terminates if and only if there exists a reduction order $\prec$ such that, for every rule $l \to r$ in R, we have $r \prec l$.

The completion procedure [78] is an algorithm that aims at converting a given set E of identities into a convergent rewriting system R such that the equational theory $\approx_E$ and the set $\{s \approx t \mid s \downarrow_R = t \downarrow_R\}$ coincide. Given a suitable reduction ordering on terms, it has been proved that the completion procedure terminates, under a fair strategy, when E contains only ground identities [83].

In the rest of this section, we assume that $\mathcal{F}$ is a disjoint union $\mathcal{F}_{AC} \uplus \mathcal{F}_U$ of AC function symbols and uninterpreted function symbols, respectively.

4.1.2 Rewriting modulo AC

In rewriting community, the set of axioms defining the AC theory is usually given by the equivalent set of identities:

$$\mathrm{AC} = \bigcup_{u \in \mathcal{F}_{AC}} \left\{ \; u(x,y) \approx u(y,x) \;,\; u(x,u(y,z)) \approx u(u(x,y),z) \; \right\}$$

Let $\approx_{AC}$ be the equational theory obtained from the set AC defined above. Given a set E of identities over Σ, it has been shown that no suitable reduction ordering allows the completion procedure to produce a convergent TRS for the union $E \cup AC$ in general. An alternative consists in in-lining AC reasoning both in the notion of rewriting step and in the completion procedure. For instance, given a rule $u(a, u(b, c))) \to t$, we would like the following reductions to be possible:

$$(1) \quad f(u(c, u(b, a)), d) \to f(t, d) \qquad\qquad (2) \quad u(a, u(c, u(d, b))) \to u(t, d)$$

Associativity and commutativity of u are needed in (1) for the subterm $u(c, u(b, a))$ to match the term $u(a, u(b, c))$, and in (2) for the term $u(a, u(c, u(d, b)))$ to be seen as $u(u(a, u(b, c)), d)$, so that the rule can be applied. More formally, this leads to the following definition of rewriting modulo AC in the ground case:

Definition 25 (Ground rewriting modulo AC)**.** *A term s rewrites to a term t modulo AC at position p by the rule $l \to r$, denoted by $s \to^{p}_{AC \setminus l \to r} t$, iff*

- (1) $s|_p =_{AC} l$ *and* $t = s[r]_p$ *or*
- (2) $l(\Lambda) = u$ *and there exists s' such that* $s|_p =_{AC} u(l, s')$ *and* $t = s[u(r, s')]_p$

In order to produce a convergent TRS, the ground AC completion procedure requires a well-founded reduction quasi-ordering $\preceq$ total on ground terms with an underlying equivalence relation which coincides with $\approx_{AC}$. Such an ordering will be called a total ground AC-reduction ordering.

4.1.3 Ground AC completion

The inference rules for ground AC completion are recalled in Figure 4.1. The rules describe the evolution of the state of a procedure, represented as a configuration $\langle\ E \mid R\ \rangle$, where E is a set of ground equations and R a ground set of rewriting rules. The initial state is $\langle\ E_0 \mid \emptyset\ \rangle$ where E_0 is a given set of ground equations.

The rule **Tri**vial removes an equation $u \approx v$ from E when u and v are equal modulo AC. **Ori**ent turns an equation into a rewriting rule according to a given total ground AC-reduction ordering $\preceq$. R is used to rewrite either side of an equation (**Sim**plify), and to reduce right hand side of rewriting rules (**Com**pose). Given a rule $l \to r$, **Col**lapse either reduces l at an inner position, or replaces l by a term smaller than r. In both cases, the reduction of l to l' may influence the orientation of the rule $l' \to r$ which is added to E as an equation in order to be re-oriented. Finally, **Ded**uce adds equational consequences of rewriting rules to E. For instance,

$$\textbf{Trivial}\ \frac{\langle\ E \cup \{\, s \approx t \,\}\ \mid\ R\ \rangle}{\langle\ E\ \mid\ R\ \rangle}\ s\ \approx_{AC}\ t$$

$$\textbf{Orient}\ \frac{\langle\ E \cup \{\, s \approx t \,\}\ \mid\ R\ \rangle}{\langle\ E\ \mid\ R \cup \{\, s \to t \,\}\ \rangle}\ t \prec s$$

$$\textbf{Simplify}\ \frac{\langle\ E \cup \{\, s \approx t \,\}\ \mid\ R\ \rangle}{\langle\ E \cup \{\, s' \approx t \,\}\ \mid\ R\ \rangle}\ s \to_{AC\setminus R} s'$$

$$\textbf{Compose}\ \frac{\langle\ E\ \mid\ R \cup \{\, l \to r \,\}\ \rangle}{\langle\ E\ \mid\ R \cup \{\, l \to r' \,\}\ \rangle}\ r \to_{AC\setminus R} r'$$

$$\textbf{Collapse}\ \frac{\langle\ E\ \mid\ R \cup \{\, g \to d, l \to r \,\}\ \rangle}{\langle\ E \cup \{\, l' \approx r \,\}\ \mid\ R \cup \{\, g \to d \,\}\ \rangle}\ \begin{cases} l \to_{AC\setminus g\to d} l' \\ g \prec l\ \vee\ (g \simeq l \wedge d \prec r) \end{cases}$$

$$\textbf{Deduce}\ \frac{\langle\ E\ \mid\ R\ \rangle}{\langle\ E \cup \{\, s \approx t \,\}\ \mid\ R\ \rangle}\ s \approx t \in \mathtt{headCP}(R)$$

Figure 4.1: Inference rules for ground AC completion

if R contains two rules of the form $u(a,b) \to s$ and $u(a,c) \to t$, then the term $u(a,u(b,c))$ can either be reduced to $u(s,c)$ or to the term $u(t,b)$. The equation $u(s,c) \approx u(t,b)$, called *critical pair*, is thus necessary for ensuring convergence of R. Head critical pairs of a set of rules are computed by the following function (a^μ stands for the maximal term w.r.t. size enjoying the assertion):

$$\mathtt{headCP}(R) = \left\{ u(b,r') \approx u(b',r) \;\middle|\; \begin{array}{l} l \to r \in R,\ \ l' \to r' \in R \\ \exists\, a^\mu :\ l =_{AC} u(a^\mu, b)\ \wedge\ l' =_{AC} u(a^\mu, b') \end{array} \right\}$$

Note that, unlike non-ground AC completion, we do not need to compute internal critical pairs in the ground case, because they are handled by the **Col**lapse rule.

Example. To get a flavor of ground AC completion, consider a modified version of the assertion given in the introduction, where the arithmetic part has been removed (and uninterpreted constant symbols renamed for the sake of simplicity)

$$\left(\begin{array}{l} u(a_1,a_4) \approx a_1\ \ \wedge\ \ a_5 \approx a_4\ \ \wedge \\ u(a_3,a_6) \approx u(a_5,a_5)\ \ \wedge\ \ a_6 \approx a_2 \end{array} \right)\ \ \Rightarrow\ \ a_1 \approx u(a_1, u(a_6,a_3))$$

1	$\mathbf{u(a_1, a_4)} \to \mathbf{a_1}$	**Ori** $u(a_1, a_4) \approx a_1$
2	$u(a_3, a_6) \to u(a_5, a_5)$	**Ori** $u(a_3, a_6) \approx u(a_5, a_5)$
3	$\mathbf{a_5} \to \mathbf{a_4}$	**Ori** $a_5 \approx a_4$
4	$u(a_3, a_6) \to u(a_4, a_4)$	**Com** 2 and 3
5	$\mathbf{a_6} \to \mathbf{a_2}$	**Ori** $a_6 \approx a_2$
6	$u(a_3, a_2) \approx u(a_4, a_4)$	**Col** 4 and 5
7	$\mathbf{u(a_4, a_4)} \to \mathbf{u(a_3, a_2)}$	**Ori** 6
8	$u(a_1, a_4) \approx u(a_1, u(a_3, a_2))$	**Ded** from 1 and 7
9	$a_1 \approx u(a_1, u(a_3, a_2))$	**Sim** 8 by 1
10	$\mathbf{u(a_1, u(a_3, a_2))} \to \mathbf{a_1}$	**Ori** 9

Figure 4.2: Ground AC completion example

The precedence $a_1 \prec_p \cdots \prec_p a_6 \prec_p u$ defines an AC-RPO ordering on terms [99] which is suitable for ground AC completion. The table in Figure 4.2 shows the application steps of the rules given in Figure 4.1 from an initial configuration $\langle\, \{u(a_1, a_4) \approx a_1, u(a_3, a_6) \approx u(a_5, a_5), a_5 \approx a_4, a_6 \approx a_2\} \mid \emptyset \,\rangle$ to a final configuration $\langle\, \emptyset \mid R_f \,\rangle$, where R_f is the set of rewriting rules $\{1, 3, 5, 7, 10\}$. It can be checked that $a_1{\downarrow_{R_f}}$ and $u(a_1, u(a_6, a_3)){\downarrow_{R_f}}$ are identical. Notice that, terms are not in canonical form modulo AC, and the rule **Tri**vial uses the symbol $\approx_{AC}$ instead of $=$.

From now on, we assume given a Shostak theory X over a signature Σ_{X}. The terms we will consider are now built from a signature Σ defined as the union of the disjoint signatures Σ_{AC}, $\Sigma_{\mathcal{E}}$ and Σ_{X}. Given a function symbol f, the notation $f \in \Sigma_i$ is an abbreviation for $f \in \mathcal{F}_i$. We also assume given a total ground AC-reduction ordering $\preceq$ defined on $\mathcal{T}_\Sigma(\mathcal{X})$ used later on for completion.

4.2 The ingredients of the combination

According to the definition of Section 2.2, the canonizer and the solver routines of a Shostak theory X only work on Σ_{X}-pure terms. The two first steps toward the combination of ground AC completion with a signature-disjoint Shostak theory X are the definition of a global canonizer for the union of the free theory of equality,

the AC theory and X and a wrapper for the solver function capable of resolving heterogeneous equations.

In this section, we first show how to construct such a canonizer. Then, we extend standard rewriting — in the spirit of normalized rewriting — to incorporate global canonization. After that, we consider the problem of solving heterogeneous equations. Finally, we pose in the last part some reasonable ordering constraints on the global canonizer and the extended solver, essential for the termination proof of our combination framework.

The definitions of the global canonizer and the solver's wrapper use a variable abstraction mechanism. Let $\alpha : \mathcal{T}_\Sigma \to \mathcal{X}$ be a global one-to-one mapping and ρ its inverse mapping. We define the variable abstraction which computes the *pure* Σ_{X}-part $[\![t]\!]$ of a heterogeneous term t as follows:

$$\begin{array}{lcll} [\![t]\!] & = & f([\![\vec{s}]\!]) & \text{when } t = f(\vec{s}) \text{ and } f \in \Sigma_{\mathsf{X}} \\ [\![t]\!] & = & \alpha(t) & \text{otherwise} \end{array}$$

Example. Given a term t of the form $2\,u(b+0, a+1-1)+b-2+3$ where u is an AC symbol and a, b are uninterpreted symbols. The abstracted value $[\![t]\!]$ of t is of the form $2\,x+y-2+3$ assuming $\alpha(u(b+0, a+1-1)) = x$ and $\alpha(b) = y$. Notice that the function $[\![\cdot]\!]$ stops at maximal Σ_{X}-aliens. The terms a and b in $u(b+0, a+1-1)$ are not abstracted.

4.2.1 Global canonization

Combining canonizers for arbitrary signature-disjoint convex theories is not very difficult. For instance, following the technique described in [81], we define a global canonizer for the union of $\mathcal{E}$, AC and X as follows:

Definition 26. *Given a signature* $\Sigma = \Sigma_{AC} \uplus \Sigma_{\mathcal{E}} \uplus \Sigma_{\mathsf{X}}$, *we define a canonizer* $\mathtt{canon}$ *which combines* $\mathtt{canon}_{AC}$ *and* $\mathtt{canon}_{\mathsf{X}}$ *by:*

$$\begin{array}{lcll} \mathtt{canon}(x) & = & x & \text{when } x \in \mathcal{X} \\ \mathtt{canon}(f(\vec{v})) & = & f(\mathtt{canon}(\vec{v})) & \text{when } f \in \Sigma_{\mathcal{E}} \\ \mathtt{canon}(u(s,t)) & = & \mathtt{canon}_{AC}(u(\mathtt{canon}(s), \mathtt{canon}(t))) & \text{when } u \in \Sigma_{\mathsf{AC}} \\ \mathtt{canon}(f_{\mathsf{X}}(\vec{v})) & = & \mathtt{canon}_{\mathsf{X}}(f_{\mathsf{X}}([\![\mathtt{canon}(\vec{v})]\!]))\rho & \text{when } f_{\mathsf{X}} \in \Sigma_{\mathsf{X}} \end{array}$$

The proofs that $\mathtt{canon}$ fulfills standard properties required for a canonizer are similar to those given in [81]. The only difference is that $\mathtt{canon}_{AC}$ is capable of

working directly on the signature Σ, which avoids the use of a variable abstraction step when canonizing a mixed AC headed term. The lemma below states that canon solves the word problem for the union of $\mathcal{E}$, AC and X.

Lemma 27. $\forall s,t \in \mathcal{T}_\Sigma, \quad s =_{\mathcal{E},AC,\mathsf{X}} t \;\Rightarrow\; \mathtt{canon}(s) \equiv \mathtt{canon}(t)$

Example. Assuming X is the theory of linear integer arithmetic and using an AC-RPO ordering based on a precedence such that $a \prec_p b$, the application of canon on $2\,u(b+0, a+1-1)+b-2+3$ yields $2\,u(a,b)+b+1$.

4.2.2 Canonized rewriting

We now extend AC rewriting with the global canonizer. From rewriting point of view, a canonizer behaves like a convergent rewriting system: it provides an effective way of computing normal forms of terms. Thus, a natural way for integrating canon in ground AC completion is to adapt normalized rewriting [90] by replacing normalization with canonization. Hence, the following definition:

Definition 28. *Let* canon *be a canonizer. We say that a term s* canon*-rewrites to a term t at position p by the rule $l \to r$, denoted by $s \rightsquigarrow^p_{l\to r} t$, iff*

$$s \to^p_{AC\setminus l\to r} t' \qquad \text{and} \qquad \mathtt{canon}(t') = t$$

The following lemma states the soundness of canonized rewriting. It can be easily proved using the definition above and the soundness of the function canon.

Lemma 29. $\forall\ s,\ t.\quad s \rightsquigarrow_{l\to r} t \Rightarrow s =_{AC,\mathsf{X},l\approx r} t$

Example. Using the rewriting rule $\gamma : u(a,b) \to b$, the term $f(b+2\,u(b,a))$ canon-rewrites to $f(3\,a)$ by $\rightsquigarrow_\gamma$ as follows:

$$\begin{array}{rcl} f(b+2\,u(b,a)) & \to_{AC\setminus\gamma} & f(b+2\,b) \\ \text{and} \quad \mathtt{canon}(f(b+2\,b)) & = & f(3\,b) \end{array}$$

Intuitively, canonized rewriting simply applies global canonization after each AC reduction step. Notice that, according to the definition, the term $f(b+2\,u(b,a))$ does not canon-rewrites to $f(3\,a)$ by $\rightsquigarrow_\delta$ using the rule $\delta : u(a+0,b) \to b$, because $u(a+0,b)$ does not match $u(b,a)$ modulo AC. Fortunately, we can easily get rid of this issue by requiring the terms manipulated in the combination framework to be in canonized form.

4.2.3 Solving heterogeneous equations

We reuse the same mappings α, ρ and the same abstraction function $[\![\cdot]\!]$ to define a wrapper `solve` for the function $\texttt{solve}_{\mathsf{X}}$ to resolve mixed equations. The definition is very simple and is given below:

Definition 30. *Given an equation $s \approx t \in \mathcal{T}_\Sigma \times \mathcal{T}_\Sigma$, we pose:*

$$\texttt{solve}(s \approx t) = \begin{cases} \bot & \text{if } \texttt{solve}_{\mathsf{X}}([\![s]\!] \approx [\![t]\!]) = \bot \\ \{\, x_i\rho \to t_i\rho \,\} & \text{if } \texttt{solve}_{\mathsf{X}}([\![s]\!] \approx [\![t]\!]) = \{x_i \approx t_i\} \end{cases}$$

4.2.4 Ordering constraints

As it is already the case for ground AC completion, we have to impose some ordering constraints on the functions `canon` and `solve` in order to prove the termination of our framework. More precisely, the global canonizer and the wrapper must be compatible with the ordering $\preceq$ used by ground AC completion, that is:

Lemma 31.

$$\begin{aligned} &\forall t \in \mathcal{T}_\Sigma, \quad \texttt{canon}(t) \preceq t \\ &\forall s,t \in \mathcal{T}_\Sigma, \quad \text{if } \texttt{solve}(s \approx t) = \bigcup\{p_i \to v_i\} \text{ then } v_i \prec p_i \end{aligned}$$

We can prove that the above properties hold when the theory X enjoys the local compatibility properties given below:

Axiom 32.

$$\begin{aligned} &\forall t \in \mathcal{T}_\Sigma, \quad \texttt{canon}_{\mathsf{X}}([\![t]\!]) \preceq [\![t]\!] \\ &\forall s,t \in \mathcal{T}_\Sigma, \quad \text{if } \texttt{solve}_{\mathsf{X}}([\![s]\!] \approx [\![t]\!]) = \bigcup\{x_i \approx t_i\} \text{ then } t_i\rho \prec x_i\rho \end{aligned}$$

In order to fulfill this axiom, the AC reduction ordering $\prec$ can be chosen as an AC-RPO ordering [99] based on a precedence relation $\prec_p$ such that $\Sigma_{\mathsf{X}} \prec_p \Sigma_{\mathcal{E}} \cup \Sigma_{AC}$. From now on, we assume that the theory X is locally compatible with $\preceq$.

Example. To solve the equation $2\,u(a,b) + b + 1 \approx 0$, we use the abstraction $\alpha = \{u(a,b) \mapsto x,\ b \mapsto y\}$ and call $\texttt{solve}_{\mathsf{X}}$ on $2\,x + y + 1 \approx 0$. Since $b \prec u(a,b)$, the solution $y \approx -2\,x - 1$ is discarded because it does not comply with the second requirement of axiom 32. The only solution which fulfills this axiom is:

$$\exists k.\ \{x \approx k\ ,\ y \approx -2\,k - 1\}$$

We then apply ρ and obtain the following set of rewriting rules:

$$\exists k.\ \{u(a,b) \to k\ ,\ b \to -2\,k - 1\}$$

Intuitively, the first solution is rejected because it would yield a rewriting rule of the form $b \to -2\,u(a,b) - 1$ which will not terminate if applied on a term containing the left-hand-side b. The existentially quantified variable k introduced by the solver can be skolemized and replaced by a fresh uninterpreted constant k_{fc} such that $k_{fc} \prec b$.

4.3 The combination framework

The definitions given in the previous section pave the way for a modular extension of the ground AC completion procedure with a Shostak theory X. In this section, we present the details of this extension and illustrate its use through an example.

4.3.1 The inference rules of AC(X)

The main idea of our combination framework is very simple. Indeed, the AC(X) algorithm is obtained by:

1. replacing the rewriting relation in AC completion with canonized rewriting;
2. replacing the equation orientation mechanism with the wrapper `solve`.

These two modifications lead to the inference rules given in Figure 4.3. The state of the procedure is a pair $\langle\ E \mid R\ \rangle$ of ground equations and ground rewriting rules. The initial configuration is of the form $\langle\ E_0 \mid \emptyset\ \rangle$ where E_0 is supposed to be a set of equations between canonized terms (*i.e.* $\texttt{canon}(E_0) = E_0$). Notice that, since AC(X)'s rules only involve canonized rewriting, the algorithm maintains the invariant that terms occurring in E and R are always in canonized forms.

The rule **Tri**vial thus removes an equation $s \approx t$ from E when s and t are syntactically equal. The rule **Ded**uce that computes AC critical pairs remains unchanged. Indeed, since the theory X provides a solver, no additional critical pairs modulo X are needed. A new rule **Bot**tom is used to detect inconsistent equations. Adding this rule is motivated by the fact that X is not necessarily an equational theory. For instance, the theory of linear arithmetic knows disequalities such as $1 \not\approx 2$. Similarly to normalized completion, localized changes are made in the

$$\textbf{Trivial}\ \frac{\langle\, E \cup \{\, s \approx t \,\} \mid R \,\rangle}{\langle\, E \mid R \,\rangle}\ s = t$$

$$\textbf{Bottom}\ \frac{\langle\, E \cup \{\, s \approx t \,\} \mid R \,\rangle}{\bot}\ \mathtt{solve}(s,t) = \bot$$

$$\textbf{Orient}\ \frac{\langle\, E \cup \{\, s \approx t \,\} \mid R \,\rangle}{\langle\, E \mid R \cup \mathtt{solve}(s,t) \,\rangle}\ \mathtt{solve}(s,t) \neq \bot$$

$$\textbf{Simplify}\ \frac{\langle\, E \cup \{\, s \approx t \,\} \mid R \,\rangle}{\langle\, E \cup \{\, s' \approx t \,\} \mid R \,\rangle}\ s \rightsquigarrow_R s'$$

$$\textbf{Compose}\ \frac{\langle\, E \mid R \cup \{\, l \to r \,\} \,\rangle}{\langle\, E \mid R \cup \{\, l \to r' \,\} \,\rangle}\ r \rightsquigarrow_R r'$$

$$\textbf{Collapse}\ \frac{\langle\, E \mid R \cup \{\, g \to d,\, l \to r \,\} \,\rangle}{\langle\, E \cup \{\, l' \approx r \,\} \mid R \cup \{\, g \to d \,\} \,\rangle}\ \begin{cases} l \rightsquigarrow_{g \to d} l' \\ g \prec l \ \vee\ (g \simeq l \wedge d \prec r) \end{cases}$$

$$\textbf{Deduce}\ \frac{\langle\, E \mid R \,\rangle}{\langle\, E \cup \{\, s \approx t \,\} \mid R \,\rangle}\ s \approx t \in \mathtt{headCP}(R)$$

Figure 4.3: The inference rules of the AC(X) algorithm

other rules. In our setting, the rule **Orient** uses the wrapper `solve` to turn equations into rewriting rules, and the other rules use the relation $\rightsquigarrow_R$ that integrates global canonization instead of $\to_{AC\setminus R}$.

4.3.2 Example

Let us now illustrate AC(X) on the example given at the beginning of this chapter.

$$\begin{array}{lll} i_1.\ \ u(a, c_2 - c_1) \approx a & \wedge & \\ i_2.\ \ u(e_1, e_2) - f(b) \approx u(d,d) & \wedge & \\ i_3.\ \ d \approx c_1 + 1 & \wedge & \\ i_4.\ \ e_2 \approx b & \wedge & \vdash \quad a \approx u(a,0) \\ i_5.\ \ u(b, e_1) \approx f(e_2) & \wedge & \\ i_6.\ \ c_2 \approx 2\, c_1 + 1 & & \end{array}$$

We apply the following strategy for processing an equality $u \approx v \in E$:

$$\mathbf{Sim}^* \ (\mathbf{Tri} \mid \mathbf{Bot} \mid (\mathbf{Ori} \ (\mathbf{Com\ Col\ Ded})^*))$$

This means that $u \approx v$ is first simplified as much as possible by **Sim**plify. Then, if it is not proven to be trivially solved by **Tri**vial or unsolvable by **Bot**tom, it is solved by **Ori**ent. Each resulting rule is added to R and then used to **Com**pose and **Col**lapse the other rules of R. Critical pairs are then computed by **Ded**uce.

1	$u(a, c_2 - c_1) \to a$	**Ori** $u(a, c_2 - c_1) \approx a$
2	$u(e_1, e_2) \to u(d, d) + f(b)$	**Ori** $u(e_1, e_2) - f(b) \approx u(d, d)$
3	$\mathbf{d \to c_1 + 1}$	**Ori** $d \approx c_1 + 1$
4	$u(e_1, e_2) \to u(c_1 + 1, c_1 + 1) + f(b)$	**Com** 2 and 3
5	$\mathbf{e_2 \to b}$	**Ori** $e_2 \approx b$
6	$u(b, e_1) \approx u(c_1 + 1, c_1 + 1) + f(b)$	**Col** 4 and 5
7	$u(b, e_1) \to u(c_1 + 1, c_1 + 1) + f(b)$	**Ori** $u(b, e_1) \approx u(c_1 + 1, c_1 + 1) + f(b)$
8	$u(c_1 + 1, c_1 + 1) + f(b) \approx f(b)$	**Sim** $u(b, e_1) \approx f(e_2)$ by 5 and 7
9	$\mathbf{u(c_1 + 1, c_1 + 1) \to 0}$	**Ori** $u(c_1 + 1, c_1 + 1) + f(b) \approx f(b)$
10	$\mathbf{u(b, e_1) \to f(b)}$	**Com** 7 and 9
11	$\mathbf{c_2 \to 2\, c_1 + 1}$	**Ori** $c_2 \approx 2\, c_1 + 1$
12	$u(a, c_1 + 1) \approx a$	**Col** 1 and 11
13	$\mathbf{u(a, c_1 + 1) \to a}$	**Ori** $u(a, c_1 + 1) \approx a$
14	$u(0, a) \approx u(a, c_1 + 1)$	**Ded** from 9 and 13
15	$u(0, a) \approx a$	**Sim** 14 by 13
16	$\mathbf{u(0, a) \to a}$	**Ori** 15

Figure 4.4: AC(X) on the running example.

The table given in Figure 4.4 shows the application of the rules of AC(X) on the example when X is instantiated by linear arithmetic. We use an AC-RPO ordering based on the precedence:

$$1 \prec_p 2 \prec_p a \prec_p b \prec_p c_1 \prec_p c_2 \prec_p d \prec_p e_1 \prec_p e_2 \prec_p f \prec_p u$$

Running AC(X) on the equalities $i_1, \ldots, i_6$ terminates and produces a convergent rewriting system $R_f = \{3, 5, 9, 10, 11, 13, 16\}$. Using R_f, we can check that a and $u(a, 0)$ canon-rewrite to the same normal form.

4.4 Correctness and termination proofs

We give in this section the detailed proofs for the correctness of AC(X). This property is stated by the theorem below and its proof is based on three intermediate theorems stating respectively soundness, completeness and termination.

Theorem 33. *Given a set E of ground equations, the application of the rules of AC(X) under a strongly fair strategy terminates and either produces $\bot$ when $E \cup AC \cup \mathsf{X}$ is inconsistent, or yields a final configuration $\langle\ \emptyset \mid R\ \rangle$ such that:*

$$\forall s, t \in \mathcal{T}_\Sigma.\ s =_{E,AC,X} t \Leftrightarrow \mathtt{canon}(s)\!\downarrow_R = \mathtt{canon}(t)\!\downarrow_R$$

As usual, in order to enforce correctness, we cannot use any (unfair) strategy. This is why we require a strongly fair one in the theorem above. It is defined as follows:

Definition 34. *A strategy is* strongly fair *when no possible application of an inference rule is infinitely delayed and* **Ori***ent is only applied over fully reduced terms.*

In the following, we shall consider a fixed run of the completion procedure

$$\langle\ E_0 \mid \emptyset\ \rangle \to \langle\ E_1 \mid R_1\ \rangle \to \ldots \to \langle\ E_n \mid R_n\ \rangle \to \langle\ E_{n+1} \mid R_{n+1}\ \rangle \to \ldots$$

starting from the initial configuration $\langle\ E_0 \mid \emptyset\ \rangle$. We denote R_∞ (resp. E_∞) the set of all encountered rules (resp. equations) and $\mathcal{R}_\omega$ (resp. E_ω) the set of persistent rules (resp. equations). More formally, these sets are defined as follows:

$$\begin{aligned} R_\infty &= \textstyle\bigcup_n R_n \\ E_\infty &= \textstyle\bigcup_n E_n \\ R_\omega &= \textstyle\bigcup_n \bigcap_{i \geq n} R_i \\ E_\omega &= \textstyle\bigcup_n \bigcap_{i \geq n} E_i \end{aligned}$$

Notice that, requiring a strongly fair strategy implies in particular that:

- $\mathtt{headCP}(R_\omega) \subseteq E_\infty$,
- $E_\omega = \emptyset$,
- R_ω is inter-reduced, that is none of its rules can be collapsed or composed by another one.

In addition, due to the assumptions made over $\texttt{canon}_{\mathsf{X}}$ and $\prec$, the following valid properties will be continuously used in the proofs:

$$\begin{array}{lllll} (P_1) & \forall t, & \texttt{canon}(t) & \preceq & t \\ (P_2) & \forall s,t, & s \simeq t & \iff & s =_{AC}\ t \\ (P_3) & \forall s,t, & s \rightsquigarrow_{R_\infty} t & \implies & t \prec s \end{array}$$

4.4.1 Soundness

The soundness property of AC(X) is ensured by the following invariant:

Theorem 35. *For any configuration* $\langle\ E_n \mid R_n\ \rangle$ *reachable from* $\langle\ E_0 \mid \emptyset\ \rangle$,

$$\forall\ s,\ t,\quad (s,t) \in E_n \cup R_n \Rightarrow s =_{AC,\mathsf{X},E_0}\ t$$

Proof. The invariant obviously holds for the initial configuration and is preserved by all the inference rules.

- The rules **Sim**plify, **Com**pose, **Col**lapse and **Ded**uce preserve the invariant since for any rule $l \to r$, if $l =_{AC,\mathsf{X},E_0}\ r$ then, for any term s rewritten into t by $\rightsquigarrow_{l\to r}$, we have $s =_{AC,\mathsf{X},E_0}\ t$.
- If **Ori**ent is used to turn an equation $s \approx t$ into a set of rules $\{p_i \to v_i\}$, by definition of solve, $p_i = x_i\rho$ and $v_i = t_i\rho$, where $\texttt{solve}_{\mathsf{X}}(\llbracket s \rrbracket \approx \llbracket t \rrbracket) = \{x_i \approx t_i\}$. By soundness of $\texttt{solve}_{\mathsf{X}}\ x_i =_{\mathsf{X},\llbracket s\rrbracket\approx\llbracket t\rrbracket}\ t_i$. An equational proof of $x_i =_{\mathsf{X},\llbracket s\rrbracket\approx\llbracket t\rrbracket}\ t_i$ can be instantiated by ρ, yielding an equational proof $p_i =_{\mathsf{X},s\approx t}\ v_i$. Since by induction $s =_{AC,\mathsf{X},E_0}\ t$ holds, we get $p_i =_{AC,\mathsf{X},E_0}\ v_i$.

□

4.4.2 Completeness

Completeness is established in several steps using a variant of the technique introduced by Bachmair *et al.* in [7] for proving completeness of completion. This technique transforms a proof between two terms which is not under a suitable form into a smaller one, and the smallest proofs are the desired ones.

The proofs we are considering are made of elementary steps, either equational steps, with AC, X and E_∞, or rewriting steps, with R_∞ and the additional (possibly infinite) rules $R_{\texttt{canon}} = \{t \to \texttt{canon}(t) \mid \texttt{canon}(t) \neq t\}$. Rewriting steps with R_∞ can be either $\rightsquigarrow_{R_\infty}$ or $\to_{R_\infty}$[1].

[1]Here,$s \longrightarrow_{R_\infty} t$ actually means $s \longrightarrow_{AC\setminus R_\infty} t'$ and $t = \texttt{canon}_{AC}(t')$.

The measure of a proof is the multiset of the elementary measures of its elementary steps. The measure of an elementary step is a 5-tuple of type

$$\texttt{multiset}(\mathcal{T}_\Sigma(\mathcal{X})) \times \mathbb{N} \times \mathbb{N} \times \mathcal{T}_\Sigma(\mathcal{X}) \times \mathcal{T}_\Sigma(\mathcal{X})$$

It takes into account the number of terms which are in a canonical form in an elementary proof: the canonical weight of a term t, $w_{\texttt{canon}}(t)$ is equal to 0 if $\texttt{canon}(t) =_{AC} t$ and to 1 otherwise. Notice that if $w_{\texttt{canon}}(t) = 1$, then $\texttt{canon}(t) \prec t$, and if $w_{\texttt{canon}}(t) = 0$, then $\texttt{canon}(t) \simeq t$. The measure of an elementary step between t_1 and t_2 is performed thanks to

- an equation is equal to $(\{\!\{t_1, t_2\}\!\}, \_, \_, \_, \_)$
- a rule $l \to r \in R_\infty$ is equal to

  $$(\{\!\{t_1\}\!\}, 1, w_{\texttt{canon}}(t_1) + w_{\texttt{canon}}(t_2), l, r)$$

  if $t_1 \rightsquigarrow_{l\to r} t_2$ (or $t_1 \to_{l\to r} t_2$), and to

  $$(\{\!\{t_2\}\!\}, 1, w_{\texttt{canon}}(t_1) + w_{\texttt{canon}}(t_2), l, r)$$

  if $t_1 \leftsquigarrow_{r\leftarrow l} t_2$ (or $t_1 \leftarrow_{r\leftarrow l} t_2$). In the case of a $\rightsquigarrow$ step, the measure is actually $(\{\!\{t_i\}\!\}, 1, w_{\texttt{canon}}(t_i), l, r)$ since the reduct is always in a canonical form.
- a rule of $R_{\texttt{canon}}$ is equal to

  $$(\{\!\{t_1\}\!\}, 0, w_{\texttt{canon}}(t_1) + w_{\texttt{canon}}(t_2), t_1, t_2)$$

  if $t_1 \to_{R_{\texttt{canon}}} t_2$, and to

  $$(\{\!\{t_2\}\!\}, 0, w_{\texttt{canon}}(t_1) + w_{\texttt{canon}}(t_2), t_2, t_1)$$

  if $t_1 \leftarrow_{R_{\texttt{canon}}} t_2$.

Elementary steps are compared lexicographically using the multiset extension of $\preceq$ for the first component, the usual ordering over natural numbers for the components 2 and 3, and $\preceq$ for last ones. Since $\preceq$ is an AC-reduction ordering, the ordering defined over proofs is well-founded.

The general methodology is to show that a proof which contains some unwanted elementary steps can be replaced by a proof with a strictly smaller measure. Since the ordering over measures is well-founded, there exists a minimal proof, and such a minimal proof is of the desired form.

Lemma 36. *A proof containing an elementary step $\longleftrightarrow_{s\approx t}$, where $s \approx t \in AC \cup \mathsf{X}$ is not minimal.*

Proof. An elementary equational step using an equation $s \approx t$ of $AC \cup \mathsf{X}$ under the context $\mathrm{C}[\_]_p$ can be reduced: the subproof

$$\mathrm{C}[s]_p \underset{s\approx t}{\longleftrightarrow} \mathrm{C}[t]_p$$

is replaced by

$$\mathrm{C}[s]_p \underset{R_{\mathrm{canon}}}{\overset{\{0,1\}}{\longrightarrow}} \mathtt{canon}(\mathrm{C}[s]_p) = \mathtt{canon}(\mathrm{C}[t]_p) \underset{R_{\mathrm{canon}}}{\overset{\{0,1\}}{\longleftarrow}} \mathrm{C}[t]_p$$

The measure strictly decreases, since for the first subproof it is equal to $\{\!\{(\{\!\{\mathrm{C}[s]_p, \mathrm{C}[t]_p\}\!\}, \_, \_, \_, \_)\}\!\}$, and for the second one, it is equal to $\{\!\{(\{\!\{\mathrm{C}[s]_p\}\!\}, \_, \_, \_, \_)^{\{0,1\}}, (\{\!\{\mathrm{C}[t]_p\}\!\}, \_, \_, \_, \_)^{\{0,1\}}\}\!\}$. The rewrite steps $\rightarrow_{R_{\mathrm{canon}}}^{\{0,1\}}$ only occur on a term which is not AC-equal to a canonical form (which is denoted by the $\{0, 1\}$ exponent). The corresponding elementary measure occurs in the global measure of the second subproof accordingly. □

Lemma 37. *A proof containing an elementary step $\longleftrightarrow_{s\approx t}$, where $s \approx t \in E_\infty$ is not minimal.*

Proof. An elementary equational step using an equation $s \approx t$ of E_∞ under the context $\mathrm{C}[\_]_p$ can be reduced. Since E_ω is empty, there is a completion state where $s \approx t$ disappears, either by **Sim**plify or **Ori**ent.

- If **Sim**plify is used to reduce s into s' by the rule $l \to r$ of R_∞, the subproof

  $$\mathrm{C}[s]_p \underset{s\approx t}{\longleftrightarrow} \mathrm{C}[t]_p$$

  is replaced by

  $$\mathrm{C}[s]_p \underset{l\to r}{\longrightarrow} \mathrm{C}[s']_p \underset{s'\approx t}{\longleftrightarrow} \mathrm{C}[t]_p$$

  The measure strictly decreases, since for the first subproof it is equal to $\{\!\{(\{\!\{\mathrm{C}[s]_p, \mathrm{C}[t]_p\}\!\}, \_, \_, \_, \_)\}\!\}$, and for the second one, it is equal to $\{\!\{(\{\!\{\mathrm{C}[s]_p\}\!\}, \_, \_, \_, \_), (\{\!\{\mathrm{C}[s']_p, \mathrm{C}[t]_p\}\!\}, \_, \_, \_, \_)\}\!\}$, and $s \succ s'$.

- If the rule **Ori**ent turns $s \approx t$ into a set of rules $\pi = \{p_i \to v_i\}$, by definition of `solve` we have $\mathtt{solve}_{\mathsf{X}}([\![s]\!] \approx [\![t]\!]) = \{x_i \approx t_i\}$ (denoted as σ) with $p_i = x_i\rho$ and $v_i = t_i\rho$. Since $\mathtt{solve}_{\mathsf{X}}$ is complete, $[\![s]\!]\sigma =_{\mathsf{X}} [\![t]\!]\sigma$. Consider a variable x of $[\![s]\!]$ or $[\![t]\!]$,

  – if $x \in \{x_i\}$ then $x\rho\pi = p_i\pi = v_i$ and $x\sigma\rho = t_i\rho = v_i$.

– if $x \notin \{x_i\}$ then $x\rho\pi = x\rho$ (since $x\rho \notin \{p_i\}$) and $x\sigma\rho = x\rho$ (since $x\sigma = x$).

In all cases, $x\rho\pi = x\sigma\rho$. The equational step using $s \approx t$ can be recovered as a compound step using π and $R_{\mathtt{canon}}$ as follows:

$$\mathrm{C}[s]_p = \mathrm{C}[[\![s]\!]\rho]_p \xrightarrow[\pi]{+} \mathrm{C}[[\![s]\!]\rho\pi]_p = \mathrm{C}[[\![s]\!]\sigma\rho]_p \xrightarrow[R_{\mathtt{canon}}]{0,1} \xleftarrow[R_{\mathtt{canon}}]{0,1} \mathrm{C}[[\![t]\!]\sigma\rho]_p = \mathrm{C}[[\![t]\!]\rho\pi]_p \xleftarrow[\pi]{+} \mathrm{C}[[\![t]\!]\rho]_p = \mathrm{C}[t]_p$$

The set of rules π belongs to R_∞, and the measure of the new subproof is a multiset containing only elements of the form $(\{\!\{\mathrm{C}[s_i]_p\}\!\}, \_, \_, \_, \_)$, where s_i is a reduct of a subterm s or t by an arbitrary number of steps of R_∞ and $R_{\mathtt{canon}}$. In any case, $\{\!\{\mathrm{C}[s_i]_p\}\!\} \prec \{\!\{\mathrm{C}[s]_p, \mathrm{C}[t]_p\}\!\}$. The new subproof is strictly smaller than the measure of the original subproof.

□

Lemma 38. *A proof containing an elementary rewriting step truly of the form $\longrightarrow_{R_\infty}$ or $\longleftarrow_{R_\infty}$ is not minimal.*

Proof. Here, each elementary step $s \longrightarrow_{R_\infty} t$ is already a $\rightsquigarrow_{R_\infty}$ step if $t = \mathtt{canon}_{AC}(t)$ is in a canonical form w.r.t canon, or it can be replaced by

$$s \underset{R_\infty}{\rightsquigarrow} \mathtt{canon}(t) \underset{R_{\mathtt{canon}}}{\longleftarrow} t$$

The measure of the first subproof is equal to $\{\!\{(\{\!\{s\}\!\}, 1, w_{\mathtt{canon}}(s) + w_{\mathtt{canon}}(t), \_, \_)\}\!\}$, and the measure of the second one is equal to $\{\!\{(\{\!\{s\}\!\}, 1, w_{\mathtt{canon}}(s), \_, \_), (\{\!\{t\}\!\}, 0, \_, \_, \_)\}\!\}$, and $t \prec s$. Since $w_{\mathtt{canon}}(t) = 1$, the measure strictly decreases.

The case $s \longleftarrow_{R_\infty} t$ is symmetrical. □

Lemma 39. *A proof containing an elementary rewriting step of the form $\rightsquigarrow_{l\to r}$ or $\leftsquigarrow_{r\leftarrow l}$, where $l \to r \in R_\infty \setminus R_\omega$ is not minimal.*

Proof. An elementary $\rightsquigarrow$ step using a rule $l \to r$ of $R_\infty \setminus R_\omega$ can be reduced. The rule $l \to r$ disappears either by **Compose** or by **Collapse**.

- If **Compose** reduces r to $r' = \mathtt{canon}(r[d])$ by the rule $g \to d$ of R_∞, the subproof

$$\mathrm{C}[l]_p \underset{l\to r}{\rightsquigarrow} \mathtt{canon}(\mathrm{C}[r]_p)$$

can be replaced by

$$\mathrm{C}[l]_p \underset{l\to r'}{\rightsquigarrow} \mathtt{canon}(\mathrm{C}[r']_p) = \mathtt{canon}(\mathrm{C}[r[d]]_p) \underset{d\leftarrow g}{\leftsquigarrow} \mathrm{C}[r]_p$$

The identity $\mathtt{canon}(\mathrm{C}[r']_p) = \mathtt{canon}(\mathrm{C}[r[d]]_p)$ holds $\mathrm{C}[r']_p$ and $\mathrm{C}[r[d]]_p$ are equal modulo $R_{\mathtt{canon}}$, that is $AC \cup \mathsf{X}$, and such terms have the same canonical forms. The measure strictly decreases, since for the first subproof it is equal to $\{\!\{(\{\!\{\mathrm{C}[l]_p\}\!\}, 1, w_{\mathtt{canon}}(\mathrm{C}[l]_p), l, r)\}\!\}$, and for the second one, it is equal to $\{\!\{(\{\!\{\mathrm{C}[l]_p\}\!\}, 1, w_{\mathtt{canon}}(\mathrm{C}[l]_p), l, r')), (\{\!\{\mathrm{C}[r]_p\}\!\}, 0, \_, \_, \_)\}\!\}$, and $r' \prec r \prec l$.

- If **Co**llapse reduces l to $l' = \mathtt{canon}(l[d])$ by the rule $g \to d$ in R_∞, the subproof

$$\mathrm{C}[l]_p \underset{l \to r}{\rightsquigarrow} \mathtt{canon}(\mathrm{C}[r]_p)$$

is replaced by

$$\mathrm{C}[l]_p \underset{g \to d}{\rightsquigarrow} \mathtt{canon}(\mathrm{C}[l[d]]_p) = \mathtt{canon}(\mathrm{C}[l']_p) \underset{R_{\mathtt{canon}}}{\longleftarrow} \mathrm{C}[l']_p \underset{l' \approx r}{\longleftrightarrow} \mathrm{C}[r]_p \underset{R_{\mathtt{canon}}}{\longrightarrow} \mathtt{canon}(\mathrm{C}[r]_p)$$

The measure strictly decreases, since for the first subproof it is equal to $\{\!\{(\{\!\{\mathrm{C}[l]_p\}\!\}, 1, w_{\mathtt{canon}}(\mathrm{C}[l]_p), l, r)\}\!\}$, and for the second one, it is equal to

$$\{\!\{(\{\!\{\mathrm{C}[l]_p\}\!\}, 1, w_{\mathtt{canon}}(\mathrm{C}[l]_p), g, d), \\ (\{\!\{\mathrm{C}[l']_p\}\!\}, \_, \_, \_, \_), (\{\!\{\mathrm{C}[l']_p\mathrm{C}[r]_p\}\!\}, \_, \_, \_, \_), (\{\!\{\mathrm{C}[r]_p\}\!\}, \_, \_, \_, \_), \}\!\}$$

The last three elements of the second multiset are strictly smaller than the element of the first multiset, since $l' \prec l$ and $r \prec l$. The first element of the second multiset is strictly smaller than the element of the first multiset, since either $g \prec l$, and the fourth component decreases, or $g \simeq l$ and $d \prec g$. In this case, $l' = d \prec r$. The first four components are identical, and the last one decreases.

The case $\rightsquigarrow$ is symmetrical. □

Lemma 40. *A proof containing a peak $s \leftarrow_{R_{\mathtt{canon}}} t \rightarrow_{R_{\mathtt{canon}}} s'$ is not minimal.*

Proof. All the terms s, t and s' involved in the peak are equal modulo AC and X, hence $\mathtt{canon}(s) = \mathtt{canon}(s')$. The subproof

$$s \leftarrow_{R_{\mathtt{canon}}} t \rightarrow_{R_{\mathtt{canon}}} s'$$

is replaced by

$$s \rightarrow^{\{0,1\}}_{R_{\mathtt{canon}}} \mathtt{canon}(s) = \mathtt{canon}(s') \leftarrow^{\{0,1\}}_{R_{\mathtt{canon}}} s'$$

The measure strictly decreases, since for the first subproof it is equal to

$$\{\!\{(\{\!\{t\}\!\}, 0, w_{\mathtt{canon}}(t) + w_{\mathtt{canon}}(s), \_, \_), (\{\!\{t\}\!\}, 0, w_{\mathtt{canon}}(t) + w_{\mathtt{canon}}(s'), \_, \_)\}\!\}$$

and for the second one, it is equal to

$$\{\!\{(\{\!\{s\}\!\}, 0, w_{\texttt{canon}}(s), \_, \_)^{\{0,1\}}, (\{\!\{s'\}\!\}, 0, w_{\texttt{canon}}(s), \_, \_)^{\{0,1\}}\}\!\}.$$

s and s' are smaller than or equivalent to t ($s, s' \preceq t$), and the second component strictly decreases, since $\texttt{canon}(s)$ and $\texttt{canon}(s')$ are in a canonical form and t is not. □

Lemma 41. *A proof containing a peak $s \leftsquigarrow_{R_\omega} t \rightsquigarrow_{R_\omega} s'$ is not minimal.*

Proof. We make a case analysis over the positions of the reductions.

- In the parallel case, the subproof

  $$s \underset{r \leftarrow l}{\overset{p}{\leftsquigarrow}} t \underset{g \to d}{\overset{q}{\rightsquigarrow}} s'$$

  can be seen as

  $$s = \texttt{canon}(t[r]_p[g]_q) \underset{R_{\text{canon}}}{\longleftarrow} t[r]_p[g]_q \underset{r \leftarrow l}{\longleftarrow} t[l]_p[g]_q \underset{g \to d}{\longrightarrow} t[l]_p[d]_q \underset{R_{\text{canon}}}{\longrightarrow} \texttt{canon}(t[l]_p[d]_q) = s'$$

  The above subproof can be replaced by

  $$s = \texttt{canon}(t[r]_p[g]_q) \overset{\{0,1\}}{\underset{R_{\text{canon}}}{\longleftarrow}} t[r]_p[g]_q \underset{g \to d}{\rightsquigarrow} \texttt{canon}(t[r]_p[d]_q) \underset{r \leftarrow l}{\leftsquigarrow} t[l]_p[d]_q \overset{\{0,1\}}{\underset{R_{\text{canon}}}{\longrightarrow}} \texttt{canon}(t[l]_p[d]_q) = s'$$

  The measure strictly decreases, since for the first subproof it is equal to $\{\!\{(\{\!\{t\}\!\}, \_, \_, \_, \_), (\{\!\{t\}\!\}, \_, \_, \_, \_)\}\!\}$, and for the second one, it is equal to

  $$\{\!\{(\{\!\{t[r]_p[g]_q\}\!\}, \_, \_, \_, \_)^{\{0,1\}}, \quad (\{\!\{t[r]_p[g]_q\}\!\}, \_, \_, \_, \_), \\ (\{\!\{t[l]_p[d]_q\}\!\}, \_, \_, \_, \_), (\{\!\{t[l]_p[d]_q\}\!\}, \_, \_, \_, \_)^{\{0,1\}}\}\!\}$$

  and both terms $t[r]_p[g]_q$ and $t[l]_p[d]_q$ are strictly smaller than $t = t[l]_p[g]_q$.

- If q is a strict prefix of p, this means that $l \to r$ can be used to collapse the rule $g \to d$, which is impossible since the strategy is strongly fair, and the application of **Co**llapse cannot be infinitely delayed.

- The case where p is a strict prefix of q is similar.

- If p and q are equal, this means that in both reductions, the extended rewriting has been used (second case of definition 25). Otherwise, again, one rule could collapse the other. This means that l and g have the same AC top function symbol u. When l and g do not share a common subterm, the reasoning is similar to the parallel case. Otherwise, if they share a common

subterm, since the strategy is fair, the head critical pair between $l \to r$ and $g \to d$ has been computed. Let a^μ the maximal common part between l and g, $l =_{AC} u(a^\mu, b)$, and $g =_{AC} u(a^\mu, b')$. The critical pair is $u(b', r) \approx u(b, d)$. The subterm $t|_p$ where both reductions occur is of the form $u(a^\mu, u(b, u(b', c)))$ (or $u(a^\mu, u(b, b'))$ if it corresponds exactly to the critical pair).

The subproof can be replaced by

$$s = \underset{R_{\text{canon}}}{\longleftarrow} t[u(u(b', r), c)]_p \underset{u(b',r)\approx u(b,d)}{\longleftrightarrow} t[u(u(b, d), c]_p \underset{R_{\text{canon}}}{\longrightarrow} s'$$

The measure strictly decreases, since for the first subproof it is equal to $\{\!\{(\{\!\{t\}\!\}, \_, \_, \_, \_), (\{\!\{t\}\!\}, \_, \_, \_, \_)\}\!\}$, and for the second one, it is equal to

$$\{\!\{(\{\!\{t[u(u(b', r), c)]_p\}\!\}, \_, \_, \_, \_), \{\!\{t[u(u(b', r), c)]_p, t[u(u(b, d), c]_p\}\!\}, \_, \_, \_, \_), \\ \{\!\{t[u(u(b, d), c]_p\}\!\}, \_, \_, \_, \_)\}\!\}$$

and both $t[u(u(b', r), c)]_p$ and $t[u(u(b, d), c]_p$ are strictly smaller than t.

□

Lemma 42. *A proof containing a peak $s \leftsquigarrow_{R_\omega} t \longrightarrow_{R_{\text{canon}}} s'$ is not minimal.*

The proof of this lemma is partly made by structural induction over t, and we need an auxiliary result in order to study how behave a proof plugged under a context.

Definition 43. *Given a context $\mathrm{C}[\bullet]_p$, and an elementary proof $\mathcal{P}$, $\mathcal{P}$ plugged under $\mathrm{C}[\bullet]_p$, denoted as $\mathrm{C}[\mathcal{P}]_p$ is defined as follows:*

- *if $\mathcal{P}$ is an equational step $s \leftrightarrow_{l\approx r} t$, $\mathrm{C}[\mathcal{P}]_p$ is $C[s]_p \leftrightarrow_{l\approx r} C[t]_p$.*
- *if $\mathcal{P}$ is a rewriting step $s \longrightarrow_{l\to r} t$, $\mathrm{C}[\mathcal{P}]_p$ is $C[s]_p \longrightarrow_{l\to r} C[t]_p$.*
- *if $\mathcal{P}$ is a rewriting step $s \rightsquigarrow_{l\to r} t$, $\mathrm{C}[\mathcal{P}]_p$ is either*

$$C[s]_p \rightsquigarrow_{l\to r} \mathtt{canon}(C[t]_p) \leftarrow^{\Lambda}_{R_{\text{canon}}} C[t]_p$$

if $C[t]_p$ is not in a canonical form, or $C[s]_p \rightsquigarrow_{l\to r} \mathtt{canon}(C[t]_p)$ otherwise.

This definition is extended to a proof made of several steps, by plugging elementary each step under the context. Notice that if a proof $\mathcal{P}$ relates two terms s and t, then $\mathrm{C}[\mathcal{P}]_p$ relates $\mathrm{C}[s]_p$ and $\mathrm{C}[t]_p$.

Lemma 44. *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two proofs which do not contain $\rightarrow_{R_\infty}$ nor $\leftarrow_{R_\infty}$. If $\mathcal{P}_1$ is strictly smaller than (resp. equivalent to) $\mathcal{P}_2$, then $\mathrm{C}[\mathcal{P}_1]_p$ is strictly smaller than (resp. equivalent to) $\mathrm{C}[\mathcal{P}_2]_p$. Moreover if $\mathcal{P}_2$ is a step $s \rightsquigarrow_{l\to r} t$, $\mathrm{C}[\mathcal{P}_1]_p$ is strictly smaller than $C[s]_p \rightsquigarrow_{l\to r} C[t]_p$*

Proof. It is enough to show the wanted result for elementary steps. Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two elementary steps such that $\mathcal{P}_1$ is strictly smaller than $\mathcal{P}_2$.

- If $\mathcal{P}_1$ and $\mathcal{P}_2$ are $\rightarrow_{R_{\mathtt{canon}}}$ steps, they are of the form

  $$s_i \underset{R_{\mathtt{canon}}}{\longrightarrow} t_i$$

  and the corresponding measures are $(\{\!\{s_i\}\!\}, 0, w_{\mathtt{canon}}(s_i) + w_{\mathtt{canon}}(t_i), s_i, t_i)$.

  - if $s_1 \prec s_2$, then $C[s_1]_p \prec C[s_2]_p$.
  - if $s_1 \simeq s_2$, and $w_{\mathtt{canon}}(s_1) + w_{\mathtt{canon}}(t_1) < w_{\mathtt{canon}}(s_2) + w_{\mathtt{canon}}(t_2)$. Since $s_1 \simeq s_2$, by the AC-totality of $\preceq$, we know that $s_1 =_{AC} s_2$, hence $w_{\mathtt{canon}}(s_1) = w_{\mathtt{canon}}(s_2)$. This means that $w_{\mathtt{canon}}(t_1) = 0$ and $w_{\mathtt{canon}}(t_2) = 1$. Hence $t_1 =_{AC} \mathtt{canon}(t_1)$, $t_1 \simeq \mathtt{canon}(t_1)$ and $t_2 \neq_{AC} \mathtt{canon}(t_2)$ and $\mathtt{canon}(t_2) \prec t_2$. Since $s_1 =_{AC} s_2$, $\mathtt{canon}(t_1) = \mathtt{canon}(t_2)$ holds, hence $t_1 \prec t_2$.

    If we look at the plugged proofs, we have $C[s_1]_p \simeq C[s_2]_p$, $w_{\mathtt{canon}}(C[s_1]_p) = w_{\mathtt{canon}}(C[s_2]_p)$, $w_{\mathtt{canon}}(C[t_1]_p) \leq w_{\mathtt{canon}}(C[t_2]_p) = 1$ and $C[t_1]_p \prec C[t_2]_p$. The measure is even on the first component, and either strictly decreases on the second component, or weakly decreases over the four first components, and strictly decreases over the last one. In all cases, $\mathrm{C}[\mathcal{P}_1]_p$ is strictly smaller than $\mathrm{C}[\mathcal{P}_2]_p$.
  - if $s_1 \simeq s_2$ and $w_{\mathtt{canon}}(s_1)+w_{\mathtt{canon}}(t_1) = w_{\mathtt{canon}}(s_2)+w_{\mathtt{canon}}(t_2)$, this means that $t_1 \prec t_2$. The case $w_{\mathtt{canon}}(t_1) = w_{\mathtt{canon}}(t_2) = 0$ is impossible, since this would imply $t_1 \simeq \mathtt{canon}(t_1) = \mathtt{canon}(t_2) \simeq t_2$. Hence $w_{\mathtt{canon}}(t_1) = w_{\mathtt{canon}}(t_2) = 1$.

    If we look at the plugged proofs, we have $C[s_1]_p \simeq C[s_2]_p$, $w_{\mathtt{canon}}(C[s_1]_p) = w_{\mathtt{canon}}(C[s_2]_p)$, $w_{\mathtt{canon}}(C[t_1]_p) = w_{\mathtt{canon}}(C[t_2]_p) = 1$ and $C[t_1]_p \prec C[t_2]_p$. The measure is even on the first four components, and strictly decreases over the last one. $\mathrm{C}[\mathcal{P}_1]_p$ is strictly smaller than $\mathrm{C}[\mathcal{P}_2]_p$.

- if $\mathcal{P}_1$ is a $\rightsquigarrow$-step, and $\mathcal{P}_2$ is a $\rightarrow_{R_{\mathtt{canon}}}$ step, necessarily, the first component strictly decreases. The measure of $\mathrm{C}[\mathcal{P}_1]_p$ is

  $$\{\!\{(\{\!\{C[s_1]_p\}\!\}, 1, w_{\mathtt{canon}}(C[s_1]_p), l_1, r_1), (\{\!\{C[t_1]\}\!\}, 0, \_, \_, \_)^{\{0,1\}}\}\!\}$$

and the measure of $\mathrm{C}[\mathcal{P}_2]_p$ is $(\{\!\{C[s_2]_p\}\!\}, 0, \_, \_, \_)$, where $t_1 \prec s_1 \prec s_2$. $\mathrm{C}[\mathcal{P}_1]_p$ is strictly smaller than $\mathrm{C}[\mathcal{P}_2]_p$.

- if $\mathcal{P}_1$ is a $\rightarrow_{R_{\mathtt{canon}}}$-step, and $\mathcal{P}_2$ is a $\rightsquigarrow$ step, necessarily, the first component weakly decreases and the second component strictly decreases.

  The measure of $\mathrm{C}[\mathcal{P}_1]_p$ is $(\{\!\{C[s_1]_p\}\!\}, 0, \_, \_, \_)$ which is strictly smaller than the measure of $C[s_2]_p \rightsquigarrow_{l_2 \rightarrow r_2} C[t_2]_p$, that is $\{\!\{(\{\!\{C[s_2]_p\}\!\}, 1, w_{\mathtt{canon}}(C[s_2]_p), l_2, r_2)\}\!\}$ since $s_1 \preceq s_2$.

- if $\mathcal{P}_1$ and $\mathcal{P}_2$ are $\rightsquigarrow$-steps, they are of the form

$$s_i \underset{l_i \rightsquigarrow r_i}{\rightsquigarrow} t_i$$

  and the corresponding measures are $(\{\!\{s_i\}\!\}, 1, w_{\mathtt{canon}}(s_i), l_i, r_i)$. The measure of $\mathrm{C}[\mathcal{P}_1]_p$ is

$$\begin{array}{l}\{\!\{(\{\!\{C[s_1]_p\}\!\}, 1, w_{\mathtt{canon}}(C[s_1]_p), l_1, r_1),\\ (\{\!\{C[t_1]_p\}\!\}, 0, w_{\mathtt{canon}}(C[t_1]), C[t_1]_p, \mathtt{canon}(C[t_1]_p))^{\{0,1\}}\}\!\}\end{array}$$

  and the measure of $C[s_2]_p \rightsquigarrow_{l_2 \rightarrow r_2} C[t_2]_p$ is $(\{\!\{C[s_2]_p\}\!\}, 1, w_{\mathtt{canon}}(C[s_2]_p), l_2, r_2)$.

  If $s_1 \prec s_2$, since $t_1 \prec s_1$, $\mathrm{C}[\mathcal{P}_1]_p$ is strictly smaller than $C[s_2]_p \rightsquigarrow_{l_2 \rightarrow r_2} C[t_2]_p$.

  Otherwise, $s_1 \simeq s_2$ and $s_1 =_{AC} s_2$. Hence $w_{\mathtt{canon}}(s_1) = w_{\mathtt{canon}}(s_2)$ and the decrease occurs on the last two components. Therefore $(\{\!\{C[s_1]_p\}\!\}, 1, w_{\mathtt{canon}}(C[s_1]_p), l_1, r_1)$ and $(\{\!\{C[t_1]_p\}\!\}, 0, w_{\mathtt{canon}}(C[t_1]_p), C[t_1]_p, \mathtt{canon}(C[t_1]_p))$ are strictly smaller than $(\{\!\{C[s_2]_p\}\!\}, 1, w_{\mathtt{canon}}(C[s_2]_p), l_2, r_2)$.

- When a step is an equational step, necessarily the decrease occurs on the first component. Since $\prec$ is compatible with plugging terms under a context, hence the wanted result.

□

We can now come to the proof of Lemma 42.

Proof. Let us denote by $l \rightarrow r$ the rule of R_ω, and $g \rightarrow d$ the rule of $R_{\mathtt{canon}}$; since l is in a canonical form (invariant of the completion run), the reduction using $g \rightarrow d$ can only take place at a position q which is above or parallel to the position p of the reduction using $l \rightarrow r$. We prove by induction that there exists a proof between s and s' which is strictly smaller than the original peak.

- In the parallel case, the subproof

$$s \underset{r\leftarrow l}{\overset{p}{\leftsquigarrow}} t \xrightarrow[g\to d]{q} s'$$

can be seen as

$$\mathtt{canon}(t[r]_p[g]_q) \underset{R_{\mathtt{canon}}}{\longleftarrow} t[r]_p[g]_q \underset{r\leftarrow l}{\longleftarrow} t[l]_p[g]_q \underset{R_{\mathtt{canon}}}{\longrightarrow} t[l]_p[d]_q$$

Notice that $t[r]_p[g]_q$ and $t[r]_p[d]_q$ are equal modulo AC,X, hence have the same canonical form. The above subproof can be replaced by

$$\mathtt{canon}(t[r]_p[g]_q) = \mathtt{canon}(t[r]_p[d]_q) \underset{R_{\mathtt{canon}}}{\longleftarrow} t[r]_p[d]_q \underset{r\leftarrow l}{\longleftarrow} t[l]_p[d]_q$$

which is actually

$$s \underset{r\leftarrow l}{\leftsquigarrow} s'$$

The measure strictly decreases, since for the first subproof it is equal to $\{\!\{(\{\!\{t\}\!\}, 1, 1, l, r), (\{\!\{t\}\!\}, \_, \_, \_, \_)\}\!\}$, and for the second one, it is equal to $\{\!\{(\{\!\{s'\}\!\}, 1, w_{\mathtt{canon}}(s'), l, r)\}\!\}$, and $s' \preceq t$.

- In the prefix case, we first prove the wanted result when the position q is equal to Λ. Now we make an induction over p, in order the establish that there is a proof between s and s', with a measure (weakly) smaller than $s{\leftsquigarrow_{r\leftarrow l}}t$, hence strictly smaller than the global measure of the peak. If $p = \Lambda$, rewriting at top with a rule of R_ω is impossible if it is not an extended rewriting, since l is in a canonical form. In the extended case, the subproof to be replaced has the form

$$\mathtt{canon}(u(r, l')) \underset{r\leftarrow l}{\leftsquigarrow} t \xrightarrow[R_{\mathtt{canon}}]{\Lambda} s'$$

where $t =_{AC} u(l, l')$, and $s' = \mathtt{canon}(u(l, l'))$. By definition of canon and since l is in a canonical form and u is an AC symbol, s' is AC-equal to $u(l, \mathtt{canon}(l'))$. The subproof can be replaced by

$$\mathtt{canon}(u(r, l')) = \mathtt{canon}(u(r, \mathtt{canon}(l'))) \underset{r\leftarrow l}{\leftsquigarrow} u(l, \mathtt{canon}(l')) =_{AC} s'$$

where the identity $\mathtt{canon}(u(r, \mathtt{canon}(l'))) = \mathtt{canon}(u(r, l'))$ holds since $u(r, \mathtt{canon}(l'))$ and $u(r, l')$ are equal modulo AC, X. The measure strictly decreases, since for the first subproof it is equal to $\{\!\{(\{\!\{t\}\!\}, 1, w_{\mathtt{canon}}(t), l, r), (\{\!\{t\}\!\}, \_, \_, \_, \_)\}\!\}$, and for the second one, it is equal to $\{\!\{(\{\!\{s'\}\!\}, 1, w_{\mathtt{canon}}(s'), l, r)\}\!\}$, and $s' \prec t$, or $s' \simeq t$ with $w_{\mathtt{canon}}(s') = w_{\mathtt{canon}}(t)$.

If p is of the form $i \cdot p'$, t is of the form $f(t_1, \ldots, t_{i-1}, t_i, t_{i+1}, \ldots, t_n)$, and the proof to be replaced

$$\mathtt{canon}(f(t_1, \ldots, t_i[r]_{p'}, \ldots, t_n)) \underset{r \leftarrow l}{\leftsquigarrow} f(t_1, \ldots, t_i[l]_{p'}, \ldots, t_n) \xrightarrow[R_{\mathtt{canon}}]{\Lambda} s'$$

We may assume without loss of generality that $t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n$ are in a canonical form, since

$$s' = \mathtt{canon}(t) = \mathtt{canon}(f(\mathtt{canon}(t_1), \ldots, \mathtt{canon}(t_{i-1}), t_i[l]_{p'}, \mathtt{canon}(t_{i+1}) \ldots, \mathtt{canon}(t_n)))$$

$$\mathtt{canon}(f(t_1, ..., t_i[r]_{p'}, ..., t_n)) = \mathtt{canon}(f(\mathtt{canon}(t_1), ..., \mathtt{canon}(t_{i-1}), t_i[r]_{p'}, \mathtt{canon}(t_{i+1})..., \mathtt{canon}(t_n)))$$

We also denote as

$$s_0 = f(t_1, \ldots, \mathtt{canon}(t_i[r]_{p'}), \ldots, t_n)$$

and

$$s'_0 = f(t_1, \ldots, \mathtt{canon}(t_i[l]_{p'}), \ldots, t_n)$$

We know that $\mathtt{canon}(t_i[l]_{p'}) \preceq t_i[l]_{p'}$, and we distinguish between two cases.

- If $\mathtt{canon}(t_i[l]_{p'}) \prec t_i[l]_{p'}$, then by induction hypothesis, there exists a proof $\mathcal{P}$ between $\mathtt{canon}(t_i[r]_{p'})$ and $\mathtt{canon}(t_i[l]_{p'})$ which is weakly smaller than

  $$\mathtt{canon}(t_i[r]_{p'}) \underset{r \leftarrow l}{\leftsquigarrow} t_i[l]_{p'}$$

  The decreasing is actually strict since an equivalent proof should be in one step, and the only possibility is a step of the form

  $$\mathtt{canon}(t_i[r]_{p'}) \underset{r \leftarrow l}{\leftsquigarrow} \mathtt{canon}(t_i[l]_{p'})$$

  However since $\mathtt{canon}(t_i[l]_{p'}) \prec t_i[l]_{p'}$ and $w_{\mathtt{canon}}(t_i[l]_{p'}) = w_{\mathtt{canon}}(t_i[l]_{p'})$ cannot be not simultaneously true, such an equivalent step is not possible. Among all possible proofs $\mathcal{P}$, we pick up a minimal one. By the previous lemmas, $\mathcal{P}$ does not contains $\rightarrow_{R_\infty}$ steps, hence $f(t_1, \ldots, \mathcal{P}, \ldots, t_n)$ is strictly smaller than

  $$\mathtt{canon}(s_0) \underset{r \leftarrow l}{\leftsquigarrow} t$$

  If we consider the proof $\mathcal{P}'$

  $$s \xleftarrow[R_{\mathtt{canon}}]{\{0,1\}} s_0 \xleftarrow{f(t_1,...,\mathcal{P},...,t_n)} s'_0 \xrightarrow[R_{\mathtt{canon}}]{\{0,1\}} s'$$

all its elementary steps are strictly smaller than $(\{\!\{t\}\!\}, 1, 1, l, r)$. We have seen that this is true for the middle part, and also for the left part $(\{\!\{s_0\}\!\}, 0, 1, s_0, s)^{\{0,1\}}$, and the right part $(\{\!\{s'_0\}\!\}, 0, 1, s'_0, s')^{\{0,1\}}$.

$\mathcal{P}'$ is a proof between s and s' which is strictly smaller than $s \leftsquigarrow_{r \leftarrow l} t$.

– If $\mathtt{canon}(t_i[l]_{p'}) \simeq t_i[l]_{p'}$, then by the AC-totality of $\preceq$, $\mathtt{canon}(t_i[l]_{p'}) =_{AC} t_i[l]_{p'}$. Since $s' = \mathtt{canon}(t)$, we know that $s' \preceq t$ and we make a case analysis:

  * If $s' \simeq t$ then s' is actually $\mathtt{canon}_{AC}(t)$ which is AC-equal to t. s' contains $t_i[l]_{p'}$ as a subterm and can be reduced with $l \to r$ to $\mathtt{canon}(s'[t_i[r]_{p'}])$ which is AC-equal to $t[t_i[r]_{p'}]_i$. Hence $\mathtt{canon}(s'[t_i[r]_{p'}]) = \mathtt{canon}(t[t_i[r]_{p'}]_i) = s$ and the proof

$$s \underset{r \leftarrow l}{\leftsquigarrow} s'$$

    is equivalent to, hence weakly smaller than $s \leftsquigarrow_{r \leftarrow l} t$.

  * If $s' \prec t$, then we can first see the peak as follows:

$$s \leftarrow^{\{0,1\}}_{R_{\mathtt{canon}}} s_0 \leftarrow_{r \leftarrow l} t \rightarrow_{R_{\mathtt{canon}}} s' = \mathtt{canon}(t)$$

    We eagerly replace every occurrence of l by r in s_0 and s', getting respectively s_1 and s''. Then s_1 and s'' are equal modulo AC and X, because any proof modulo AC and X between t and s' can be replayed by replacing the σ-instances of AC and X used originally by σ'-instances where $x\sigma'$ is $x\sigma$ where every occurrence of l is replaced by r. We get the new proof

$$s \underset{R_{\mathtt{canon}}}{\overset{\{0,1\}}{\leftarrow}} s_0 \underset{l \to r}{\overset{*}{\longrightarrow}} s_1 \underset{R_{\mathtt{canon}}}{\overset{\{0,1\}}{\longrightarrow}} \mathtt{canon}(s_1) = \mathtt{canon}(s'') \underset{R_{\mathtt{canon}}}{\overset{\{0,1\}}{\leftarrow}} s' = \mathtt{canon}(t)$$

    Since $s' \prec t$, all terms in the above proof are strictly smaller than t, hence the measure of this proof is strictly smaller than $(\{\!\{t\}\!\}, 1, 1, l, r)$.

If the proof occurs under a context $t[\bullet]_q$, we know that there is a proof $\mathcal{P}$ between $s = \mathtt{canon}(t[r]_{q \cdot p'})$ and $\mathtt{canon}(t)$ which is weakly smaller than $(\{\!\{t[l]_{q \cdot p'}\}\!\}, 1, 1, l, r)$ (case $\rightarrow_{R_{\mathtt{canon}}}$ at Λ). Hence

$$s \overset{\mathcal{P}}{\leftrightarrow} \mathtt{canon}(t) \underset{R_{\mathtt{canon}}}{\overset{\{0,1\}}{\leftarrow}} s'$$

is a proof between s and s' which is weakly smaller than

$$\{\!\{(\{\!\{t[l]_{q \cdot p'}\}\!\}, 1, 1, l, r), (\{\!\{s'\}\!\}, 0, 1, s', \mathtt{canon}(t))^{\{0,1\}}\}\!\}$$

whereas the measure of the original peak is

$$\{\!\{(\{\!\{t\}\!\}, 1, 1, l, r), (\{\!\{t\}\!\}, 0, 2, t, s')\}\!\}$$

Since $s' \preceq t$, the measure of the new proof is strictly smaller than the measure of the original peak.

□

Theorem 45. *If s and t are two terms such that $s \longleftrightarrow^*_{AC,\mathsf{X},E_\infty,R_\infty} s'$ then $\mathtt{canon}(s)\!\downarrow_{R_\omega} = \mathtt{canon}(t)\!\downarrow_{R_\omega}$.*

Proof. If s and s' are equal modulo $\longleftrightarrow^*_{AC,\mathsf{X},E_\infty,R_\infty}$, so are $\mathtt{canon}(s)$ and $\mathtt{canon}(s')$. By the above lemmas, a minimal proof between $\mathtt{canon}(s)$ and $\mathtt{canon}(s')$ is necessary of the form $\mathtt{canon}(s)(\rightsquigarrow_{R_\omega} \cup \rightarrow_{R_{\mathtt{canon}}})^*(\leftsquigarrow_{R_\omega} \cup \leftarrow_{R_{\mathtt{canon}}})^*\mathtt{canon}(s')$. This sequence of steps can also be seen as $\mathtt{canon}(s) \rightarrow^*_{R_{\mathtt{canon}}} (\rightsquigarrow_{R_\omega} \rightarrow^*_{R_{\mathtt{canon}}})^* (\leftarrow^*_{R_{\mathtt{canon}}} \leftsquigarrow_{R_\omega})^* \leftarrow^*_{R_{\mathtt{canon}}} \mathtt{canon}(s')$. By definition $\rightarrow_{R_{\mathtt{canon}}}$ cannot follow a $\rightsquigarrow_{R_\omega}$-step, and $\mathtt{canon}(s)$ and $\mathtt{canon}(s')$ cannot be reduced by $\rightarrow_{R_{\mathtt{canon}}}$, hence the wanted result. □

4.4.3 Termination

The proof of termination partly reuses some facts used for the termination proof of AC-ground completion (based on Higman's lemma), but also needs some intermediate lemmas which are specific to our framework[2]. We shall prove that, under a strongly fair strategy, R_ω is finite and obtained in a finite time (by cases on the head function symbol of the rule's left-hand side), and then we show that R_ω will clean up the next configurations and the completion process eventually halts on $\langle\ \emptyset \mid R_\omega\ \rangle$. In order to make our case analysis on rules, and to prove the needed invariants, we define several sets of terms (assuming without loss of generality that $E_0 = \mathtt{canon}(E_0)$):

$$\begin{aligned}
T_0 &= \{t \mid \exists t_0, e_1, e_2 \in \mathcal{T}_\Sigma(\mathcal{X}), e_1 \approx e_2 \in E_0 \text{ and } t_0 = e_i|_p \text{ and } t_0 \rightsquigarrow^*_{R_\infty} t\}\\
T_{0\mathsf{X}} &= T_0 \cup \{f_{\mathsf{X}}(t_1, \ldots, t_n) \mid f_{\mathsf{X}} \in \Sigma_{\mathsf{X}} \text{ and } \forall i, t_i \in T_{0\mathsf{X}}\}\\
T_1 &= \{t \mid t \in T_0 \text{ and } \forall p, t|_p \in T_{0\mathsf{X}}\}\\
T_2 &= \{u(t_1, \ldots, t_n) \mid 2 \leq n \text{ and } u \in \Sigma_{AC} \text{ and } \forall i, t_i \in T_1\}
\end{aligned}$$

T_0 is the set of all terms and subterms in the original problem as well as their reducts by R_∞. The set $T_{0\mathsf{X}}$ moreover contains terms with X-aliens in T_0. T_1 is the set of terms that can be introduced by X from terms of T_0 (by solving or canonizing). T_2 is a superset of the terms built by critical pairs.

[2] We assume that $\bot$ is not encountered, otherwise, termination is obvious.

Lemma 46. $\forall \gamma, t, s,\ \gamma \in R_\infty \cap T_j^2 \wedge t \in T_i \wedge t \rightsquigarrow_\gamma s \Rightarrow s \in T_i$, *for* $i, j = 1, 2$.

The proof is by structural induction over terms (for dealing with rewriting under a context) and by case analysis over T_i when rewriting at the top level. It uses the (quasi-immediate) fact that $T_0 \cap T_2 \subseteq T_1$.

Lemma 47. *For all accessible configuration* $\langle\ E_n \mid R_n\ \rangle$, $E_n \cup R_n \subseteq T_1^2 \cup T_2^2$.

The proof is by induction over n, and uses Lemma 46.

The first step of the termination proof is to show that $R_\omega \cap T_1^2$ is finite (Lemma 50). It is specific to our framework, due to the presence of X[3].

Lemma 48. *Under a strongly fair strategy, if* $l \to r_n$ *is created at step* n *in* R_n *and* $l \to r_m$ *at step* m *in* R_m*, with* $n < m$*, then* r_m *is a reduct of* r_n *by* $\rightsquigarrow_{R_\infty}$.

Proof. The proof is by induction over the length of the derivation, and by case analysis over the rule which has been applied.

- **Ori**ent applied on $s = t$ cannot create a new rule $p \to v$ with an already present left hand side, because the strongly fair strategy implies that s and t are fully reduced, and the new left hand side p is a subterm of s or t.
- **Sim**plify, **Col**lapse and **Ded**uce do not create a new rule.
- **Com**pose obviously preserves the invariant.

□

Corollary 49. *Under a strongly fair strategy,* R_∞ *is finitely branching.*

Proof. If R_∞ is not finitely branching, there exist an infinite sequence of rules $(l \to r_n)_n$ where $l \to r_n$ first appears in $\langle\ E_n \mid R_n\ \rangle$. Thanks to Lemma 48, since R_∞ is included in $\prec$, the sequence $(r_n)_n$ is strictly decreasing w.r.t $\prec$. The well-foundedness of $\prec$ contradicts the infinity of $(r_n)_n$. □

Lemma 50. *Under a strongly fair strategy, the set of rules in* $\mathcal{R}_\omega \cap T_1^2$ *is finite.*

Proof. If $l \to r$ belongs to the set $\mathcal{R}_\omega \cap T_1^2$, l is reduct of a term l_0 in E_0 by $\rightsquigarrow_{R_\infty}$. Since $\rightsquigarrow_{R_\infty}$ is terminating (because it is included in $\prec$), and finitely branching (above corollary), any term has finitely many reducts by $\rightsquigarrow_{R_\infty}$. In particular since E_0 is finite, there are finitely many possible left-hand side. Moreover since in R_ω two distinct rules have distinct left-hand sides, $\mathcal{R}_\omega \cap T_1^2$ is finite. □

[3] X may change the head function symbol of terms in an equational proof, which is not the case of AC in standard ground AC completion.

Here is the second step of the termination proof, finiteness of $R_\omega \cap T_2^2$, which is mostly the same as in the usual AC-ground completion:

Lemma 51. *The set of persistent rules in $\mathcal{R}_\omega$ which are in T_2^2 is finite.*

Proof. The set $R_\omega \cap T_2^2$ can be divided into a finite union of sets, according to the top AC function symbol of the left hand-side of the rules. We shall prove that for each $u \in \Sigma_{AC}$, the corresponding subset is finite.

Let u be a fixed AC function symbol, and let $u(l_1, \ldots, l_n) \to r$ be a rule of $R_\omega \cap T_2^2$. By definition of T_2, and by the soundness of R_∞, each l_i is equal modulo ACX, E_0 to a term l_i^0 in E_0. Since l_i is irreducible by R_ω (otherwise the rule $u(l_1, \ldots, l_n) \to r$ would have collapsed), there is a rewriting proof $l_i \leftsquigarrow_{R_\omega}^* l_i^0$. Notice that two distinct rules in R_ω have some distinct left-hand sides (otherwise one would have collapsed the other) (this implies in particular that R_ω is finitely branching). Since $\rightsquigarrow_{R_\omega}$ is included in a well-founded ordering, and is finitely branching any term has a finite number of reducts. Since E_0 is finite, each l_i belongs to the *finite* set of reducts $Red(E_0)$ of E_0 by $\rightsquigarrow_{R_\omega}$. By Higman's lemma, if there is an *infinite* number of rules where the left-hand side is of the form $u(t_1, \ldots, t_n)$, there exist two rules $l \to r$ and $l' \to r'$, such that the multiset of arguments $\{\!\{l_1, \ldots, l_n\}\!\}$ of l is included in the multiset of arguments $\{\!\{l'_1, \ldots, l'_m\}\!\}$ of l'. This would imply that the second rule collapses by the first one, which contradicts its persistence. Hence the wanted result. □

When R_ω has been proven to be finite, we show that once it is obtained, R_ω will "clean up" the configuration within a finite number of steps, hence the termination:

Theorem 52. *Under a strongly fair strategy, AC(X) terminates.*

Proof. When the strategy is strongly fair, R_ω is finite. Moreover each rule in R_ω is obtained within a finite number of steps. Once all persistent rules are present in the rules of the configuration $\langle\ E \mid R\ \rangle$, the rule **Orient** always returns an empty set of rules. If the measure of a configuration is the triple made of the number of remaining critical pairs to generate, the multiset of terms in R (compared with $\prec$), and the number of equations on E, it strictly decreases. □

4.5 Variable abstraction and multiset ordering

Although AC-RPO orderings are suitable when proving termination of completion procedures or of term rewriting systems modulo AC in general, they are not easily

implementable in practice and can be a source of inefficiency. We show in this chapter that a simple variable abstraction preprocessing step *à la* Bachmair et al. allows AC(X) to use a partial multiset reduction ordering instead of a full AC-RPO ordering. The principle of this step is similar to the **Ext**ension inference rule found in Abstract Congruence Closure [6]. In order to enable the use of such ordering as an input for AC(X), we have to transform the original set of ground equations E to a simpler one containing only *abstracted* equations. Let K be a set of constant symbols disjoint from Σ and $\mathcal{X}$ and $\prec_{\mathsf{X}}$ be a total rewrite ordering on $\mathcal{T}_{\Sigma_X \cup K}$. We define two sets of terms $\mathcal{T}_\emptyset$ and $\mathcal{T}_{AC}$ as follows:

$$\mathcal{T}_\emptyset \quad = \quad \left\{ f(v_1, \cdots, v_n) \;\middle|\; \begin{array}{ll} f \in \Sigma_\emptyset & \wedge \\ arity(f) = n & \wedge \\ \bigwedge_{i=1}^{n} v_i \in \mathcal{T}_{\Sigma_{\mathsf{X}} \cup K} & \end{array} \right\}$$

$$\mathcal{T}_{AC} \quad = \quad \left\{ u(v_1, u(v_2, \ldots, u(v_{n-1}, v_n)\ldots)) \;\middle|\; \begin{array}{ll} u \in \Sigma_{AC} & \wedge \\ n \geq 2 & \wedge \\ \bigwedge_{i=1}^{n} v_i \in \mathcal{T}_{\Sigma_{\mathsf{X}} \cup K} & \end{array} \right\}$$

Definition 53 (Abstracted equations)**.** *An equation $s \approx t$ is said to be abstracted if one of the following statements holds:*

1. $s,\ t \in \mathcal{T}(\Sigma_{\mathsf{X}} \cup K)$
2. $s \in (\mathcal{T}_\emptyset \cup \mathcal{T}_{AC})$ *and* $t \in \mathcal{T}(\Sigma_{\mathsf{X}} \cup K)$
3. $s,\ t \in \mathcal{T}_{AC}$ *and* $s(\Lambda) = t(\Lambda)$

The set of all abstracted equations is denoted by $\mathcal{A}$.

Let π be an abstraction function from $\mathcal{T}_{AC} \cup \mathcal{T}_\emptyset$ to K such that if $\pi(s) = \pi(t)$ then $s =_{AC,\mathsf{X}} t$. Given a set E^0 of ground equations, the term abstraction of E^0 consists in applying, as long as possible, the following inference rules, starting from the initial configuration $\langle\ E^0 \mid \emptyset\ \rangle$.

$$\textbf{Ab}\text{stract1}\ \frac{\langle\ E \uplus \{s \approx t\} \mid E_{\mathcal{A}}\ \rangle}{\langle\ E \mid E_{\mathcal{A}} \cup \{s \approx t\}\ \rangle}\ s \approx t \in \mathcal{A}$$

$$\textbf{Ab}\text{stract2}\ \frac{\langle\ E \cup \mathcal{C}[f(\vec{v})] \approx t \mid E_{\mathcal{A}}\ \rangle}{\langle\ E \cup \mathcal{C}[k] \approx t \mid E_{\mathcal{A}} \cup \{f(\vec{v}) \approx k\}\ \rangle}\ \mathcal{C}[f(\vec{v})] \approx t \notin \mathcal{A}$$

where,

1. $f(\vec{v}) \in (\mathcal{T}_{\emptyset} \cup \mathcal{T}_{AC})$
2. $k = \pi(f(\vec{v}))$

The propositions 54 and 55 state respectively the termination and the correctness of the abstraction process.

Proposition 54. *The application of the rules **Ab**stract**1** and **Ab**stract**2** terminates and produces a configuration of the form $\langle\ \emptyset \mid E_{\mathcal{A}}^{\infty}\ \rangle$, where $E_{\mathcal{A}}^{\infty} \subseteq \mathcal{A}$.*

Proof. The proof of termination is immediate using a decreasing measure. The size of a configuration is equal to the total sum of the sizes of the terms in its first component. Here, the size of a term is recursively defined in a standard way with 1 for the size of constants in K, and 2 for the size of other constants.

It remains to show that if a configuration is of the form $\langle\ E \mid E_{\mathcal{A}}\ \rangle$ and $E \neq \emptyset$, at least one rule applies. Let $s \approx t$ be an equation in E. If $s \approx t \in \mathcal{A}$ the **Ab**stract**1** applies. Otherwise, since $s \approx t \notin \mathcal{A}$, by condition 1. of Definition 53, there is a minimal subterm of s or t which does not belong to $\mathcal{T}_{\Sigma_{\mathsf{X}} \cup K}$. This term thus has a suitable form to fulfill condition 1. in the rule **Ab**stract**2** which applies. □

Proposition 55. *Let $\langle\ E^0 \mid \emptyset\ \rangle \to^* \langle\ \emptyset \mid E_{\mathcal{A}}^{\infty}\ \rangle$ be a fixed run of the abstraction process. For any terms $s, t \in \mathcal{T}_{\Sigma}$, we have:*

$$s =_{E^0,\mathsf{X},AC} t \iff s =_{E_{\mathcal{A}}^{\infty},\mathsf{X},AC} t$$

Proof. The direction $\Rightarrow$ is immediate for **Ab**stract**1**. For **Ab**stract**2**, it rests on the fact that a step using $\mathcal{C}[f(\vec{v})] \approx t$ can be replaced by two steps, the first one using $f(\vec{v}) \approx k$ and the second one using $\mathcal{C}[k] \approx t$.

In order to prove $\Leftarrow$, we use the following invariant: if $\langle\ E \mid E_{\mathcal{A}}\ \rangle \to \langle\ E' \mid E'_{\mathcal{A}}\ \rangle$, $s =_{E',E'_{\mathcal{A}},\mathsf{X},AC} t$ and s and t do not contain any constant in K, then $s =_{E,E_{\mathcal{A}},\mathsf{X},AC} t$. This is immediate when the rule **Ab**stract**1** is applied. When **Ab**stract**2** replaces

$\mathcal{C}[f(\vec{v})] \approx t$ by $\{f(\vec{v}) \approx k, \mathcal{C}[k] \approx t\}$, we first replace every step using $\mathcal{C}[k] \approx t$ by a compound step using $\mathcal{C}[k] \approx \mathcal{C}[f(\vec{v})]$ followed by $\mathcal{C}[f(\vec{v})] \approx t$. Then all occurrences of k are replaced by $f(\vec{v})$ in intermediate terms, and the now useless steps using $f(\vec{v}) \approx f(\vec{v})$ (former $f(\vec{v}) \approx k$) are removed. The transformed proof is now in $=_{E,E_{\mathcal{A}},\mathsf{X},AC}$, and since neither s nor t contain constants in K, they are not affected by these transformations. □

Once we have shown how to abstract the initial set of equations E, we will define the reduction ordering $\prec$ that we will use in AC(X). We do not need this ordering to be total on the terms in $\mathcal{T}_{\Sigma_{\mathsf{X}} \cup K} \cup \mathcal{T}_{\emptyset} \cup \mathcal{T}_{AC}$. We only need a partial reduction ordering which allows us to get well oriented rewriting rules from the abstracted equations.

Definition 56. *Let $\prec_{\mathsf{X}}^{mset}$ be the multiset extension of $\prec_{\mathsf{X}}$. Our reduction ordering is defined by:*

1. $\forall v_1, v_2 \in \mathcal{T}_{\Sigma_{\mathsf{X}} \cup K},\ \ v_1 \prec_{\mathsf{X}} v_2 \Rightarrow v_1 \prec v_2$
2. $\mathcal{T}_{\Sigma_{\mathsf{X}} \cup K} \prec \mathcal{T}_{\emptyset}$
3. $\mathcal{T}_{\Sigma_{\mathsf{X}} \cup K} \prec \mathcal{T}_{AC}$
4. $\forall u(\vec{v}_1), u(\vec{v}_2) \in \mathcal{T}_{AC},\ \ \{\{\vec{v}_1\}\} \prec_{\mathsf{X}}^{mset} \{\{\vec{v}_2\}\} \Rightarrow u(\vec{v}_1) \prec u(\vec{v}_2)$

After that, we have to show that AC(X) does not introduce non-abstracted equations when collapsing rules, computing critical pairs, using canonized rewriting, and solving equations. Hence, the following lemma:

Lemma 57. *For any configuration $\langle\ E_n^{\infty} \mid R_n\ \rangle$ reachable from $\langle\ E_{\mathcal{A}}^{\infty} \mid \emptyset\ \rangle$, we have:*

$$\forall (s,t) \in (E_n^{\infty} \cup R_n), \qquad s \approx t \in \mathcal{A}$$

Proof. The lemma obviously holds for the initial state. For the induction step, we can easily show that the abstracted form of equations is preserved by canonized rewriting wrt an abstracted rule, hence so as when applying the inference rules **Sim**plify, **Com**pose and **Col**lapse. Concerning **Ded**uce, we notice by inspecting the definition of `headCP`, that when $l \to r$ and $l' \to r'$ are abstracted oriented equations, so is the resulting critical pair. The only subtle case is **Ori**ent, in particular when solving an equation $s \approx t$, with $s \in \mathcal{T}(\Sigma_{\mathsf{X}} \cup K)$ and $t \in \mathcal{T}_{\emptyset} \cup \mathcal{T}_{AC}$. Due to the definition of $\prec$ and to the fact that the solver has to fulfill the ordering constraints stated in Axiom 32, the solution of $s \approx t$ has to be $t \mapsto s$. □

Now, the last thing to do is to prove that the $\prec$ ordering defined above is a suitable ordering for the AC(X) completion procedure. This is actually the case since, on the equations in $\mathcal{A}$, it coincides with the AC-RPO ordering based on a precedence $\prec_p$ such that $\Sigma_{\mathsf{X}} \prec_p K \prec_p \Sigma_{\mathcal{E}} \cup \Sigma_{AC}$, and we know by Lemma 57 that AC(X) will only manipulates equations in $\mathcal{A}$ if initialized with abstracted equations.

4.6 Implementation and evaluation

4.6.1 Implementation

We implemented the AC(X) algorithm as well as a preprocessing step that enables the use of a partial multiset reduction ordering at the heart of the Alt-Ergo theorem prover. The technical details of the implementation are given in Chapter 5. Currently, X can be instantiated by the theories of linear integer arithmetic, linear rational arithmetic, records, bit-vectors and enumerated datatypes. As described in Section 4.3, the state of the procedure is a pair $\langle\ E \mid R\ \rangle$ of equations and rules. Again, we apply the following strategy for processing an equality $u \approx v \in E$:

$$\mathbf{Sim}^*\ (\mathbf{Tri} \mid \mathbf{Bot} \mid (\mathbf{Ori}\ (\mathbf{Com}\ \mathbf{Col}\ \mathbf{Ded})^*))$$

As in the example illustrating a run of AC(X), $u \approx v$ is first simplified as much as possible by **Sim**plify. Then, if it is not proven to be trivially solved by **Tri**vial or unsolvable by **Bot**tom, it is solved by **Ori**ent. Each resulting rule is added to R and then used to **Com**pose and **Col**lapse the other rules of R. Critical pairs are then computed by **Ded**uce.

The implementation of AC(X) is incremental *i.e.* is able to process new equalities on the fly. It is backtrackable for free thanks to persistent data structures. It also produces explanations: each encountered inconsistency is explained by a subset of the assumed facts leading to this inconsistency.

4.6.2 Experimental results

We benchmark AC(X) and compare its performances with our own SMT solver Alt-Ergo [30] and some state-of-the-art solvers (Z3 v2.8, CVC3 v2.2, Simplify v1.5.4). All measures are obtained on a laptop running Linux equipped with a 2.58*GHz* dual-core Intel processor and with 4*Gb* main memory. Provers are given a time limit of five minutes for each test and memory limitation is managed by the system. The results are given in seconds; we write TO for *timeout* and OM for *out of memory*.

Our test suite is made of crafted *ground* formulas which are valid in the combination of the theory of linear arithmetic LA, the free theory of equality $\mathcal{E}$ and a small part of the theory of sets defined by the symbols $\cup$, $\subseteq$, the singleton constructor $\{\cdot\}$, and the following axioms:

$$\mathcal{S}_\cup \left\{ \begin{array}{rrl} \mathit{Assoc}: & \forall x,y,z. & x \cup (y \cup z) \approx (x \cup y) \cup z \\ \mathit{Commut}: & \forall x,y. & x \cup y \approx y \cup x \end{array} \right.$$

$$\mathcal{S}_\subseteq \left\{ \begin{array}{rrl} \mathit{SubTrans}: & \forall x,y,z. & x \subseteq y \ \wedge \ y \subseteq z \ \Rightarrow \ x \subseteq z \\ \mathit{SubSuper}: & \forall x,y,z. & x \subseteq y \ \Rightarrow \ x \subseteq y \cup z \\ \mathit{SubUnion}: & \forall x,y,z. & x \subseteq y \ \Rightarrow \ x \cup z \subseteq y \cup z \\ \mathit{SubRefl}: & \forall x. & x \subseteq x \end{array} \right.$$

Figure 4.5: Axiomatization of a small part of the theory of sets

The theories $\mathcal{E}$ and LA are built-in for all SMT solvers we use for our experiments. However, contrarily to AC(X) which also natively handles associativity and commutativity, SMT solvers use a generic mechanism for instantiating the axioms $\mathcal{S}_\cup$ to reason modulo the AC properties of $\cup$.

In order to get the most accurate information about AC(X), we first benchmark a stand-alone version of our algorithm on ground formulas that can be proved without $\mathcal{S}_\subseteq$. In a second step, we consider ground formulas that are only provable with some axioms in $\mathcal{S}_\subseteq$. Since these axioms are not directly handled by AC(X), we benchmark a modified version of ALT-ERGO (to benefit from its instantiation mechanism) with AC(X) as its core decision procedure.

In the following, we use the standard mathematical notation $\bigcup_{i=1}^{d} a_i$ for the terms of the form $a_1 \cup (a_2 \cup (\cdots \cup a_d)) \cdots)$ and we write $\bigcup_{i=1}^{d} a_i ; b$ for terms of the form $a_1 \cup (a_2 \cup (\cdots \cup (a_d \cup b))) \cdots)$.

Benchmark of a stand-alone AC(X)

We consider two categories of formulas. The first category C_1 is of the form

$$\bigwedge_{p=1}^{n} (\{e\} \cup \bigcup_{i=1}^{d} a_i^p) \approx b^p \ \Rightarrow \ \underbrace{\bigwedge_{p=1}^{n-1} \bigwedge_{q=p+1}^{n} \ \bigcup_{i=d}^{1} a_i^p ; b^q \approx \bigcup_{i=d}^{1} a_i^q ; b^p}_{G}$$

and the second category C_2 is of the form

$$\bigwedge_{p=1}^{n} (\{t_p - p\} \cup \bigcup_{i=1}^{d} a_i^p) \approx b^p \wedge \bigwedge_{p=1}^{n-1} t_p + 1 \approx t_{p+1} \Rightarrow G$$

Notice that n is the number of hypothesis equations and d is the maximal depth of AC terms.

Proving the validity of C_1-formulas only requires the theory $\mathcal{E}$ and the AC properties of the union symbol. These formulas are directly provable by AC($\emptyset$) and the results for this instance are given in the first column of the table in Figure 4.6. In order to prove C_1-formulas with SMT solvers, the axioms in $\mathcal{S}_\cup$ have to be put in their context. The last four columns of the table contain the results for ALT-ERGO, Z3, CVC3 and SIMPLIFY.

n, d	AC($\emptyset$)	ALT-ERGO	Z3	CVC3	SIMPLIFY
3, 3	**0.01**	0.19	0.22	0.40	0.18
3, 6	**0.01**	32.2	OM	132	OM
3, 12	**0.01**	TO	OM	OM	OM
6, 3	**0.01**	11.2	1.10	13.2	2.20
6, 6	**0.02**	TO	OM	OM	OM
6, 12	**0.02**	TO	OM	OM	OM
12, 3	**0.16**	TO	5.64	242	11.5
12, 6	**0.24**	TO	OM	OM	OM
12, 12	**0.44**	TO	OM	OM	OM

Figure 4.6: The results for category C_1

In order to prove the validity of C_2-formulas, the theory $\mathcal{E}$, the AC properties of $\cup$ and the theory of linear arithmetic LA are required. These ground formulas are directly provable by AC(LA) and the results are given in the first column of the table in Figure 4.7. Similarly to category C_1, the last four columns of the table contain the results for the SMT solvers we considered. Again, the axioms $\mathcal{S}_\cup$ have to be provided in the context, whereas linear arithmetic is directly handled by the built-in decision procedures of these provers.

Benchmark of ALT-ERGO with AC(X)

We now analyze the performances of AC(X) when it is used as the core decision procedure of ALT-ERGO. For that, we consider a third category C_3 of formulas of

n, d	AC(LA)	ALT-ERGO	Z3	CVC3	SIMPLIFY
3, 3	**0.01**	1.10	0.03	0.11	0.19
3, 6	**0.01**	TO	3.67	4.21	OM
3, 12	**0.01**	TO	OM	OM	OM
6, 3	**0.02**	149	0.10	2.26	2.22
6, 6	**0.02**	TO	17.7	99.3	OM
6, 12	**0.04**	TO	OM	OM	OM
12, 3	**0.27**	TO	0.35	44.5	11.2
12, 6	**0.40**	TO	76.7	TO	OM
12, 12	**0.72**	TO	OM	OM	OM

Figure 4.7: The results for category C_2

the form

$$\bigwedge_{p=1}^{n} \bigcup_{i=1}^{d} \{e_i^p\} \approx b^p \wedge \bigcup_{i=1}^{d} \{e + e_i^p\} \approx c^p \wedge e \approx 0 \Rightarrow \bigwedge_{p=1}^{n} c^p \subseteq (b^p \cup \{e_d^p\}) \cup \{e\}$$

Proving the validity of C_3-formulas requires the theory $\mathcal{E}$, the AC properties of $\cup$, the theory of linear arithmetic LA and additionally some axioms in $\mathcal{S}_{\subseteq}$. We thus *only* provide the axioms $\mathcal{S}_{\subseteq}$ in the context of the modified version of ALT-ERGO, whereas *all* the axioms in $\mathcal{S}_{\subseteq}$ and $\mathcal{S}_{\cup}$ are given in the context of the other SMT solvers. The results of this category are given in Figure 4.8.

Benchmarks analysis

The results in Figures 4.6 and 4.7 show that, contrary to the axiomatic approach, built-in AC reasoning is little sensitive to the depth d of terms: given a fixed number n of equations, the running time is proportional to d. However, we notice a slowdown when n increases. This is due to the fact that AC(X) has to process a quadratic number of critical pairs generated from the equations in the hypothesis. From Figure 4.8, we remark that ALT-ERGO with AC(X) performs better than the other provers. The main reason is that its instantiation mechanism is not spoiled by the huge number of intermediate terms the other provers generate when they instantiate the AC axioms.

n, d	ALT-ERGO with AC(LA)	ALT-ERGO	Z3	CVC3	SIMPLIFY
3, 3	**0.02**	3.16	0.09	10.2	OM
3, 6	**0.04**	TO	60.6	OM	OM
3, 12	**0.12**	TO	OM	OM	OM
6, 3	**0.07**	188	0.18	179	OM
6, 6	**0.12**	TO	TO	OM	OM
6, 12	**0.66**	TO	OM	OM	OM
12, 3	**0.20**	TO	0.58	OM	OM
12, 6	**0.43**	TO	TO	OM	OM
12, 12	**1.90**	TO	OM	OM	OM

Figure 4.8: The results for category C_3

4.6.3 Instantiation issues

Although AC(X) is effective on ground formulas, its integration as the core decision procedure of ALT-ERGO suffers from a *bad interaction* between the built-in treatment of AC and the axiom instantiation mechanism of ALT-ERGO which is roughly done as follows:

- each axiom of the form $\forall \bar{x}.\, \mathcal{F}(\bar{x})$ provided in the context comes with a pattern P (also called *trigger*) which consists of a set of subterms of $\mathcal{F}$ that covers $\bar{x}$;
- the solver maintains a set G of *known* terms extracted syntactically from the *ground* literals that occur during its proof search;
- G is partitioned into a set of equivalence classes according to the ground equalities currently known by the solver;
- new *ground* formulas $\mathcal{F}\sigma$ are generated by matching P against G modulo the equivalence classes.

Let us show how this mechanism is used to prove the following ground formula:

$$(F_1) \qquad (e \approx d \cup a \ \wedge\ b \subseteq d \ \wedge\ c \approx a \cup d) \ \Rightarrow\ b \cup a \subseteq c$$

For that, we only need to use the *SubUnion* axiom defined in Section 4.5:

$$\mathit{SubUnion}: \qquad \forall x, y, z.\ x \subseteq y \ \Rightarrow\ x \cup z \subseteq y \cup z$$

Let us assume that the pattern for this axiom is the term $x \cup z \subseteq y \cup z$. This pattern is matched against the term $b \cup a \subseteq c$ by looking for a substitution σ such that $(x \cup z \subseteq y \cup z)\sigma = b \cup a \subseteq c$ modulo the set of equivalence classes $\{\{e, d \cup a, a \cup d, c\},\ \{a\},\ \{b\},\ \{d\},\ \{b \cup a\},\ \{b \subseteq d\},\ \{b \cup a \subseteq c\}\}$. Such a substitution exists and maps x to b, z to a and y to d since the term c is in the same class as $d \cup a$. The proof of F_1 follows from the ground instance $b \subseteq d \Rightarrow b \cup a \subseteq d \cup a$ of *SubUnion*.

Let us now explain the limitation of the interaction between AC(X) and the instantiation mechanism. The hypothesis $e \approx d \cup a$ is useless (from a logical point of view) to prove $b \cup a \subseteq c$. Hence, the following formula F_2 is equivalent to F_1:

$$(F_2) \qquad (b \subseteq d\ \wedge\ c \approx a \cup d)\ \Rightarrow\ b \cup a \subseteq c$$

However, the cooperation of Alt-Ergo and AC(X) fails to prove F_2. The reason is that, since the term $d \cup a$ does not syntactically occur in F_2, the equivalence classes are just $\{\{a \cup d, c\},\ \{a\},\ \{b\},\ \{d\},\ \{b \cup a\},\ \{b \subseteq d\},\ \{b \cup a \subseteq c\}\}$ and the matching algorithm fails to match $x \cup z \subseteq y \cup z$ against $b \cup a \subseteq c$.

4.7 Prospective extensions

We have shown in the previous section that AC(X) is an efficient algorithm for reasoning in the union of the free theory of equality, the AC theory and an arbitrary signature disjoint Shostak theory X that fulfills some reasonable ordering requirements. In this section, we report on three relevant extensions that we have explored. We discuss for each of them the issues we have encountered and some possible solutions that we plan to investigate in the future in order to get rid of them.

4.7.1 Matching modulo AC and a set of ground equations

As shown in Section 4.6.3, extending Alt-Ergo's axioms instantiation mechanism modulo AC is mandatory to fully integrate AC(X) as its core decision procedure. Consequently, we considered the combination of E-matching[4] and AC-matching to enable this integration. However, we realized — after a quick investigation — that

[4] *a.k.a.* matching modulo ground equalities

one could not design an E-AC-matching algorithm which combines E-matching and AC-matching and that would terminate for any input. In this section, we give a simple example which validates this affirmation and discuss a possible solution to get rid of it.

Let T be a theory. Let t be a ground term and p a term. Matching t against p modulo T, denoted $p \leq^{?}_{\mathrm{T}} t$, consists in finding the set sbt of substitutions such that:

$$\forall \sigma_i \in sbt. \quad \left\{ \begin{array}{lll} 1. & t =_{\mathrm{T}} p\sigma_i & \wedge \\ 2. & \forall x \in \mathcal{D}om(\sigma_i).\ x \notin \mathcal{V}ars(p) \Rightarrow x\sigma_i = x & \wedge \\ 3. & \forall \sigma_j \in sbt.\ i \neq j \Rightarrow \forall x \in \mathcal{D}om(\sigma_i) \cap \mathcal{D}om(\sigma_j).\ \neg(x\sigma_i =_{\mathrm{T}} x\sigma_j) & \end{array} \right.$$

The first condition makes t and $p\sigma_i$ equal modulo T. The second one forces the inclusion $\mathcal{D}om(\sigma_i) \subseteq \mathcal{V}ars(p)$. The third one eliminates redundancies modulo T.

The problem of matching modulo a given set of ground equalities E is defined by instantiating T with the equational theory of E. AC-Matching is obtained by instantiating T with the AC theory. The E-AC-matching problem is defined by instantiating T with a combination of the equational theory of a set of ground equalities and the AC theory.

We say that E-matching (resp. AC-matching) is *finitary*, because every matching problem modulo a set E of ground equalities (resp. modulo AC) admits a finite set sbt of solutions. However, E-AC-matching is *infinitary*. For instance, the following problem where $E = \{\ u(a,b) = a\ \}$, $u \in \Sigma_{AC}$ and $x,\ y$ are variables

$$u(x, u(y, y)) \leq^{?}_{AC,E} a$$

has an infinite number of solutions. In fact, the set below is a subset of sbt.

$$\bigcup_{k \in \mathbb{N}} \left\{ x \mapsto a,\ \ y \mapsto \underbrace{u(b, u(b, u(\ldots)))}_{term\ of\ depth\ k} \right\}$$

Future works

We intend to explore alternative approaches in the future in order to overcome this issue. For instance, this could consist in using AC-narrowing [121] and context-free grammars to generate finite sets of substitutions schemes. We could then enumerate the substitutions incrementally while interacting with the SAT solver and the decision procedures. This would ensure a form of fairness between the different parts of the SMT-solver and the search process would progress.

4.7.2 Reasoning in presence of non-linear multiplication

The theory of linear arithmetic handles multiplication by constants — which stands for repeated additions — but not non-linear multiplication. However, we can at least reason modulo the AC properties of non-linear multiplication using the AC(X) framework. This allows us, for instance, to prove the following formula, where $*$ denotes the non-linear multiplication symbol

$$(a * b) - 2 \approx \gamma \ \wedge \ a * c \approx \beta \ \vdash \ (\gamma + 2) * c \approx \beta * b \tag{4.1}$$

In fact, consider the AC*(LA$_\mathrm{S}$) instance of AC(X), where $\Sigma_{AC} = \{*\}$ and X is instantiated with the Shostak part of linear arithmetic. The AC terms $a * b$ and $a * c$ are considered as aliens by LA$_\mathrm{S}$. Consequently, the two first equations simply yield the rewriting rules $a * b \to \gamma + 2$ and $a * c \to \beta$. The conclusion is then deduced by a straightforward critical pair computation.

Nevertheless, AC*(LA$_\mathrm{S}$) fails to prove the formulas 4.2 and 4.3 given below. The distributivity axiom of multiplication over addition is required to prove theses examples. But, it is not handled neither by LA$_\mathrm{S}$ nor by AC completion. Note that, we also need to know that $2 * c$ is a linear term and equal to $2 \cdot c$ to discharge the formula 4.3, where " $\cdot$ " stands for multiplication by constants.

$$(a * b) - 2 \approx \gamma \ \wedge \ a * c \approx \beta \ \vdash \ \gamma * c + 2 * c \approx \beta * b \tag{4.2}$$

$$(a * b) - 2 \approx \gamma \ \wedge \ a * c \approx \beta \ \vdash \ \gamma * c + 2 \cdot c \approx \beta * b \tag{4.3}$$

It is quite frustrating to realize that AC*(LA$_\mathrm{S}$) is not able to discharge simple formulas involving distributivity. In order to get rid of this limitation, we investigated the design of a new procedure, called ACD*(LA$_\mathrm{S}$), that strengthen the AC*(LA$_\mathrm{S}$) instance in order to handle this property and to allow non-linear terms to become linear when possible. We were able to achieve this strengthening via a minor extension of global canonization.

The new canonizer $\texttt{canon}_{\mathrm{D}^*}$ that we plugged in the AC*(LA$_\mathrm{S}$) instance to obtain ACD*(LA$_\mathrm{S}$) is defined as follows:

$$
\begin{array}{lcll}
\mathtt{canon}_{\mathtt{D}^*}(x) & = & x & \text{when } x \in \mathcal{X} \\
\mathtt{canon}_{\mathtt{D}^*}(f(\vec{v})) & = & f(\mathtt{canon}_{\mathtt{D}^*}(\vec{v})) & \text{when } f \in \Sigma_{\mathcal{E}} \\
\mathtt{canon}_{\mathtt{D}^*}(f_{\mathsf{X}}(\vec{v})) & = & \mathtt{canon}_{\mathsf{X}}(f_{\mathsf{X}}([\![\mathtt{canon}_{\mathtt{D}^*}(\vec{v})]\!]))\rho & \text{when } f_{\mathsf{X}} \in \Sigma_{\mathsf{X}} \\
\mathtt{canon}_{\mathtt{D}^*}(u(t_1,t_2)) & = & \mathtt{canon}_{AC}(u(t_1',t_2')) & \text{when } u \in \Sigma_{\mathsf{AC}} \setminus \{*\} \\
\multicolumn{4}{l}{\quad \text{where} \quad t_i' = \mathtt{canon}_{\mathtt{D}^*}(t_i)}
\end{array}
$$

$$
\begin{array}{lcll}
\mathtt{canon}_{\mathtt{D}^*}(t_1 * t_2) & = & \mathtt{canon}(t') & \\
 & & \text{where} \quad \{\!\{s_1,\ldots,s_n\}\!\} = \mathcal{A}_{\{*\}}(\mathtt{canon}_{\mathtt{D}^*}(t_1)) \cup \mathcal{A}_{\{*\}}(\mathtt{canon}_{\mathtt{D}^*}(t_2)) & \\
 & & \text{and} \quad \kappa = \mathcal{A}_{\{+\}}(s_1) \times \ldots \times \mathcal{A}_{\{+\}}(s_n) & \\
 & & \text{and} \quad t' = \textstyle\sum_{\tau \in \kappa} \mathtt{monomial}(\tau) &
\end{array}
$$

where the function `monomial` simply builds the corresponding monomial from a given tuple. For instance, applying this function on $(-3,\ -b,\ a,\ b,\ 2)$ is expected to return the term $(-3)\cdot 2\cdot(-1)\cdot b*a*b$. The constant (-1) that appears in the result comes form the element $(-b)$ of the tuple. Notice that the new canonizer coincides with `canon` except when treating a non-linear term. In this case, we additionally simulate the distributivity axiom. Let us clarify the mechanism underlying this simulation through a simple example.

Example 58. *The term $t = (2\cdot a + b - 3) * (a - b + 2)$ is canonized as follows:*

1. *The subterms of t are already in canonical form with respect to $\mathtt{canon}_{\mathtt{D}^*}$. Therefore, we have $s_1 = 2\cdot a + b - 3$ and $s_2 = a - b + 2$.*

2. *Since $[\cdot]$ stand for repeated additions, The aliens parts of s_1 and s_2 with respect to $+$ are given by: $\mathcal{A}_+(s_1) = \{\!\{a, a, b, -3\}\!\}$ and $\mathcal{A}_+(s_2) = \{\!\{a, -b, 2\}\!\}$.*

3. *We then compute the cartesian product as follows:*

$$
\kappa \ = \ \left\{
\begin{array}{cccccccc}
(a,a) & , & (a,a) & , & (a,b) & , & (a,-3) & , \\
(-b,a) & , & (-b,a) & , & (-b,b) & , & (-b,-3) & , \\
(2,a) & , & (2,a) & , & (2,b) & , & (2,-3) &
\end{array}
\right\}
$$

4. *After that, we compute the sum from the tuples in κ:*

$$\begin{array}{rcccccccccc} & a*a & + & a*a & + & a*b & + & (-3)\cdot a & + \\ t' = & (-1)\cdot b*a & + & (-1)\cdot b*a & + & (-1)\cdot b*b & + & (-1)\cdot(-3)\cdot b & + \\ & 2\cdot a & + & 2\cdot a & + & 2\cdot b & + & 2\cdot(-3) & \end{array}$$

5. *Finally, we compute the canonical form of t' w.r.t* `canon` *and obtain that:*

$$\texttt{canon}_{\mathrm{D}^*}(t) = 2\cdot a*a - a*b - b*b + a + 5\cdot b - 6$$

Termination issues

We noticed that ACD*(LA$_\mathrm{S}$) is not guaranteed to terminate for any input when LA$_\mathrm{S}$ reason modulo linear arithmetic over integers. For instance, solving the equation $2\cdot(a*a) \approx a$ in this framework would loop indefinitely. Indeed,

1. solving an equality of the form $2^j\cdot(a_i*a_i) \approx a_i$ (where j is an integer constant) will yield a set of two rules $\{\, a_i * a_i \to a_{i+1}\ ,\ a_i \to 2^j\cdot a_{i+1} \,\}$, where a_{i+1} is a fresh existential variable;

2. applying the second rule on the left-hand-side of the first one will collapse it. Consequently, the equality $(2^j\cdot a_{i+1})*(2^j\cdot a_{i+1}) \approx a_{i+1}$ has to be replayed;

3. simplifying this equality as much as possible will yield $2^{2j}\cdot(a_{i+1}*a_{i+1}) \approx a_{i+1}$ which is solved similarly to the equality in 1. and the process loops indefinitely.

Future works

In the near future, we plan to study whether ACD*(LA$_\mathrm{S}$) is correct and terminating when LA$_\mathrm{S}$ reasons modulo linear arithmetic over rationals. For the integer case, we intend to explore the possibility of constraining the application of the distributivity axiom in order to ensure the termination of our framework.

4.7.3 Extending AC(X) with a first order rewriting system

The combination of the ground AC completion procedure with Shostak theories was a first step towards the integration of rewriting techniques in SMT. In a second step, we considered the extension of AC(X) with a user-defined rewriting system.

For instance, we were interested in reasoning modulo the rewriting system made of the following rule

$$m(p(x,y),z) \;\to\; p(m(x,z),m(y,z))$$

where m, p are two AC function symbols, and x, y, z are universally-quantified variables. This rule encodes the distributivity property of m over p.

Normalized completion [90] is a framework that extends the AC completion procedure with a user-defined first-order convergent rewriting system. Our AC(X) framework can be seen as an adaptation of *ground* normalized completion, when the rewriting system is equivalent to a Shostak theory X. Consequently, in order to extend AC(X) with a rewriting system, we explored the integration of Shostak theories in normalized completion. In this section, we discuss the issues we have encountered during our investigations.

Experimental extension of normalized completion

Figure 4.9 shows the "experimental extension" of normalized completion with a Shostak theory. The main changes made in the original inference rules are:

1. the replacement of the relation $\downarrow_S$ by $\nleftrightarrow_S$ in the side-conditions of **Ori**ent and **Nor**malize. The relation $\rightsquigarrow_S$ is expected to be the adaptation of canonized rewriting for the non-ground case, since the rules in S may contain variables.

2. the use of a new rewriting relation, denoted $\bullet\!\!\rightsquigarrow_{R/S}$, instead of normalized rewriting in the side-conditions of **Sim**plify, **Com**pose and **Col**lapse. This new relation is expected to be the extension of normalized rewriting with global canonization.

3. the adaptation of the functions Θ and Ψ that compute *normalizing pairs* and the function `CP` that computes *critical pairs* to take the theory X into account.

A. Extending the rewriting relations. Once, the inference rules "extended", we tried to formally define canonized rewriting for the non-ground case (denoted $\rightsquigarrow$) and canonized-normalized rewriting (denoted $\bullet\!\!\rightsquigarrow$). The first relation should take into account the variables that may occur in terms when rewriting modulo AC. We came up with the following definition:

$$\textbf{Orient}\ \frac{\langle\, E \cup \{\, s \approx t \,\} \mid R \,\rangle}{\langle\, E \cup \Theta(s,t,\mathsf{X}) \mid R \cup \Psi(s,t,\mathsf{X}) \,\rangle}\ \begin{cases} \mathtt{solve}(s,t) \neq \bot \wedge \\ s = s\!\downarrow_S \ \wedge\ t = t\!\downarrow_S \end{cases}$$

$$\textbf{Deduce}\ \frac{\langle\, E \mid R \,\rangle}{\langle\, E \cup \{\, s \approx t \,\} \mid R \,\rangle}\ s \approx t \in \mathtt{CP}(R, \mathsf{X}, \mathcal{T})$$

$$\textbf{Normalize}\ \frac{\langle\, E \cup \{\, s \approx t \,\} \mid R \,\rangle}{\langle\, E \cup \{\, s' \approx t' \,\} \mid R \,\rangle}\ s' = s\!\downarrow_S \ \wedge\ t' = t\!\downarrow_S$$

$$\textbf{Trivial}\ \frac{\langle\, E \cup \{\, s \approx t \,\} \mid R \,\rangle}{\langle\, E \mid R \,\rangle}\ s = t$$

$$\textbf{Simplify}\ \frac{\langle\, E \cup \{\, s \approx t \,\} \mid R \,\rangle}{\langle\, E \cup \{\, s \approx t' \,\} \mid R \,\rangle}\ t \bullet\!\!\rightsquigarrow_{R/S} t'$$

$$\textbf{Compose}\ \frac{\langle\, E \mid R \cup \{\, l \to r \,\} \,\rangle}{\langle\, E \mid R \cup \{\, l \to r' \,\} \,\rangle}\ r \bullet\!\!\rightsquigarrow_{R/S} r'$$

$$\textbf{Collapse}\ \frac{\langle\, E \mid R \cup \{l \to r \,\} \,\rangle}{\langle\, E \cup \{\, l' \approx r \,\} \mid R \,\rangle}\ l \bullet\!\!\rightsquigarrow_{R/S} l' \ \wedge l \neq l'$$

$$\textbf{Bottom}\ \frac{\langle\, E \cup \{\, s \approx t \,\} \mid R \,\rangle}{\bot}\ \mathtt{solve}(s,t) = \bot$$

Figure 4.9: Extending normalized completion with a Shostak theory X

Definition 59. *Let* canon *be a canonizer. A term s* canon*-rewrites to a term t at position p by the rule $l \to r$ if there exists a substitution σ such that:*

$$s \to^{\sigma,p}_{AC \setminus l \to r} t' \qquad \text{and} \qquad \mathtt{canon}(t') = t$$

We denote this reduction by $s \rightsquigarrow^{\sigma,p}_{l \to r} t$ (or simply by $s \rightsquigarrow t$ when it is not ambiguous).

The second relation interleaves S-normalization and canonized rewriting. We defined it follows:

Definition 60. *Let* canon *be a canonizer and $\rightsquigarrow$ its corresponding canonized rewriting relation. Let S be an AC-convergent rewriting system. A term s S-*canon*-rewrites to a*

term t at position p by the rule $l \to r$ if there exists a substitution σ such that:

$$s' = s\{\!\!\{_S \qquad \text{and} \qquad s' \rightsquigarrow^{\sigma,p}_{l\to r} t$$

We denote this reduction by $s \bullet\!\!\rightsquigarrow^{\sigma,p}_{l\to r/S} t$ or simply by $s \bullet\!\!\rightsquigarrow t$.

B. Normalizing and critical pairs. Contrary to normalized completion, we have in our setting additional function symbols that are interpreted in the theory X. Reasoning modulo X is required when computing normalizing pairs and critical pairs. This requirement clearly appears when computing complete sets of unifiers. Thus, we extended the definitions of Θ and Ψ as follows:

$$\left\{ \begin{array}{lcl} \Psi(s,t,\mathsf{X}) & = & \mathtt{solve}(s,t) \\ \Theta(s,t,\mathsf{X}) & = & \bigcup_{\rho_1 \in \Psi(s,t,\mathsf{X})} \bigcup_{\rho_2 \in S} (\Theta_1(\rho_1,\rho_2) \cup \Theta_2(\rho_1,\rho_2)) \end{array} \right.$$

with

$$\Theta_1(g \to d, l \to r) = \left\{ g\sigma[r\sigma]_q \approx d\sigma \;\middle|\; \begin{array}{ll} q \in \mathtt{FPos}_{\mathtt{AC}}(g) & \wedge \\ \sigma \in \mathtt{CSU}_{\mathtt{AC},\mathsf{X}}(g|_q, l) & \end{array} \right\}$$

$$\Theta_2(g \to d, l \to r) = \left\{ l\sigma[d\sigma]_q \approx r\sigma \;\middle|\; \begin{array}{ll} q \in \mathtt{FPos}_{\mathtt{AC}}(l) & \wedge \\ q \neq \Lambda & \wedge \\ \sigma \in \mathtt{CSU}_{\mathtt{AC},\mathsf{X}}(l|_q, g) & \end{array} \right\}$$

where $\mathtt{CSU}_{\mathsf{T}}$ denotes complete sets of unifiers modulo T and $\mathtt{FPos}_{\mathtt{AC}}$ denotes all the positions, modulo AC, of a given term. Note that, the equalities we solve in this new framework may contain variables. Thus, the wrapper of $\mathtt{solve}_{\mathsf{X}}$ should be able to deal with terms with universally quantified variables.

Similarly, we adapted critical pairs computation as follows, to integrate X:

$$\mathtt{CP}(R,\, \mathsf{X},\, \mathcal{T}) \;=\; \bigcup \left(\begin{array}{ll} \mathtt{CP}^{=\Lambda}(l_1 \to r_1,\, l_2 \to r_2,\, \mathsf{X},\, \mathcal{T}) & \cup \\ \mathtt{CP}^{\neq\Lambda}(l_1 \to r_1,\, l_2 \to r_2,\, \mathsf{X},\, \mathcal{T}) & \cup \\ \mathtt{CP}^{\neq\Lambda}(l_2 \to r_2,\, l_1 \to r_1,\, \mathsf{X},\, \mathcal{T}) & \end{array} \;\middle|\; \begin{array}{l} l_1 \to r_1 \in R \\ l_2 \to r_2 \in R \\ \Lambda(l_1) = u \in \Sigma_{AC} \\ \Lambda(l_2) = u \in \Sigma_{AC} \end{array} \right)$$

with

$$\mathtt{CP}^{=\Lambda}(l_1 \to r_1,\ l_2 \to r_2,\ \mathsf{X},\ \mathcal{T}) \quad = \quad \left\{ \begin{array}{rcl|l} r_1\sigma & \approx & r_2\sigma, & \sigma \in \mathtt{CSU}_{\mathcal{T},\mathsf{X}}(l_1, l_2) \\ r_1\sigma & \approx & u(r_2, y)\sigma, & \\ u(r_1, x)\sigma & \approx & r_2\sigma, & \\ u(r_1, x)\sigma & \approx & u(r_2, y)\sigma & \end{array} \right\}$$

and

$$\mathtt{CP}^{\neq\Lambda}(l_1 \to r_1,\ l_2 \to r_2,\ \mathsf{X},\ \mathcal{T}) \quad = \quad \left\{ \begin{array}{rcl|ll} r_1\sigma & \approx & l_1[r_2]_q\sigma, & q \in \mathtt{FPos}_{\mathtt{AC}}(l) & \wedge \\ & & & q \neq \Lambda & \wedge \\ r_1\sigma & \approx & l_1[u(r_2, y)]_q\sigma & \sigma \in \mathtt{CSU}_{\mathcal{T},\mathsf{X}}(l_1|_q, l_2) & \end{array} \right\}$$

Some encountered issues

A. Unification and matching issues. The experimental framework above requires the combination of unification algorithms for the Shostak theory X and the AC theory (*resp.* a theory $\mathcal{T}$ between AC and AC$\cup S$). This combination is mandatory when computing complete sets of unifiers in the definitions of normalizing pairs and critical pairs. Unfortunately, combining unification algorithms is known to be a hard task. For instance, unification in the combination of the AC theory and the theory of linear arithmetic remains an open problem.

Beside that, the combination of matching algorithms is also required. Indeed, the terms we manipulate in our setting are mixture of interpreted, uninterpreted and AC symbols and may contain variables. Therefore, the substitutions used for rewriting in definitions 59 and 60 have to be computed using pattern matching techniques in the combination of the AC theory and the theory X, and eventually in presence of uninterpreted function symbols.

Example 61. *Let u, v be two AC symbols and a, b two uninterpreted functions symbols. Assume that $R = \{v(2 \cdot z + a, a - 1) \to v(b - 2 \cdot z + x, 0\}$ and $S = \{u(a, a + y) \to v(a - 1, a + y + 1)\}$. Consider the instantiation of X with the theory of linear rational arithmetic. The term $s = u(u(a, b), a + x - 1)$ S-*canon*-rewrites to $t = v(b, 0)$ as follows:*

- *the substitution $\sigma_1 = \{y \mapsto x - 1\}$ is a solution of the matching problem $u(a, a + y) \leq^?_{AC,\mathsf{X}} u(a, a - 1 + x)$. Therefore $s\downarrow_S = v(a - 1, a + x)$,*

- *the substitution* $\sigma_2 = \{z \to \frac{1}{2} \cdot x\}$ *is a solution of the matching problem* $v(2 \cdot z + a, a - 1) \leq^?_{AC,\mathsf{X}} v(a - 1, a + x)$. *Therefore,* $v(a - 1, a + x) \rightsquigarrow_R^{\sigma_2,\Lambda} v(b, 0)$.

B. Issues related to the solver. Even if the initial set E_0 of equations is ground, the functions Θ and `CP` may introduce non-ground equations that are not (easily) solvable by the wrapper `solve`. The example below illustrates this assertion.

Example 62. *Let* f, g, a, b, c *be uninterpreted function symbols. Assume that* $E_0 = \{g(c) \approx c\}$ *and* $S = \{f(x, y, g(c)) \to f(x, a, a) + f(y, b, b) + f(x, y, c)\}$. *Solving the equation* $g(c) \approx c$ *simply yields the rewriting rule* $g(c) \to c$. *This rule is then used by* Θ *to compute normalizing pairs. Consequently, the equation* $f(x, y, c) \approx f(x, a, a) + f(y, b, b) + f(x, y, c)$, *which simplifies to* $0 \approx f(x, a, a) + f(y, b, b)$, *is introduced by* Θ_2. *However, it cannot be directly solved because* $\mathcal{V}ars(f(x, a, a)) \notin \mathcal{V}ars(-f(y, a, a))$ *and* $\mathcal{V}ars(f(y, a, a)) \notin \mathcal{V}ars(-f(x, a, a))$. *A possible solution of this issue is to introduce a fresh function symbol* h *of arity* 2 *and to transform the equations into the rules* $\{f(x, a, a) \to h(x, y)\ ,\ f(y, a, a) \to -h(x, y)\}$.

4.8 Related and future works

Related works

AC completion has been studied for a long time in the rewriting community [84, 102]. A generic framework for combining completion with a generic built-in equational theory E has been proposed in [72]. Normalized completion [90] is designed to use a modified rewriting relation when the theory E is equivalent to the union of the AC theory and a convergent rewriting system $\mathcal{S}$. In this setting, rewriting steps are only performed on $\mathcal{S}$-normalized terms. AC(X) can be seen as an adaptation of ground normalized completion to efficiently handle the theory E when it is equivalent to the union of the AC theory and a Shostak theory X. In particular, $\mathcal{S}$-normalization is replaced by the application of the canonizer of X. This modular integration of X allows us to reuse proof techniques of ground AC completion [89] to show the correctness of AC(X).

Tiwari [120] efficiently combined equality and AC reasoning in the Nelson-Oppen framework. Kapur [74] used ground completion to demystify Shostak's congruence closure algorithm and Bachmair *et al.* [6] compared its strategy with other ones into an abstract congruence closure framework. While the latter approach can also handle AC symbols, none of these works formalized the integration of Shostak theories into ground (AC) completion.

Future works

As illustrated in Section 4.6.3, the main concern for using AC(X) as a core decision procedure in ALT-ERGO is that it does not saturate equivalent classes of known ground terms modulo AC. A naive (and incomplete) solution would consist in adding, for each known ground AC-term t, a few number of AC equivalent terms (for instance by bounding the length of the AC equational proof between them). We rather plan to investigate a more elaborate solution which would consist in extending the pattern-matching algorithm of ALT-ERGO to exploit both ground equalities and properties of AC symbols. Additional prospective extensions are discussed in more details in Section 4.7. We also plan to extend AC(X) to handle the AC theory with unit or idempotence. This will be a first step towards a decision procedure for a substantial part of the finite sets theory.

CHAPTER 5

Combination of Decision Procedures in ALT-ERGO

In this chapter, we describe the techniques used at the core of ALT-ERGO to enable reasoning in a union of several background theories, such as linear arithmetic over rationals and integers, the free theory of equality, the theory of arrays, the theory of enumerated data types and the AC theory.

The general architecture of the prover, given in Figure 5.1, is similar to that of Figure 1.1. It is composed of a preprocessor part, a SAT solver engine, a "decision procedures" component and a "matching modulo ground equalities" module.

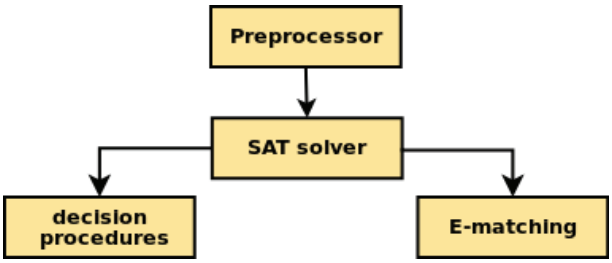

Figure 5.1: The general architecture ALT-ERGO

Reasoning in a combination of theories is handled by the "decision procedures" part. This component constitutes the core of the SMT solver. Its architecture is shown in Figure 5.2. It is built upon two combination approaches:

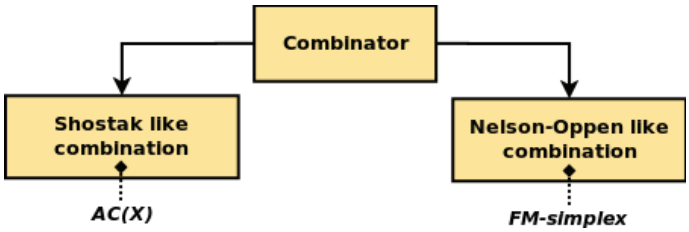

Figure 5.2: The simplified architecture ALT-ERGO's core

1. **Shostak-like combination approach**: This method handles the combination of theories that are *equational* and *convex*. These include the free theory of equality, the AC theory, and the equational parts of the theories of linear arithmetic and enumerated data types. In particular, AC(X) fits in this component.

2. **Nelson-Oppen like combination approach**: This technique is used to combine the theories that do not fit in the first approach, such as the theory of arrays, the inequalities of linear arithmetic and the non-convex part of enumerated data types. In particular, the FM-simplex procedure (except for equalities resolution) is implemented in this component.

On top of that, the communication between these frameworks is delegated to the Combinator module.

We show in the next sections the interfaces of these components and explain how they are working in practice. Note that, the presentation is done "bottom-up" in order to describe the interfaces of modules in the bottom of the hierarchy before those on its top.

5.1 The Shostak-like combination framework

Figure 5.3 shows the architecture of the modules that constitute our Shostak-like combination approach. It consists in extending the union-find data structure (UFX) of a congruence closure algorithm (CCX) to reason in the union of convex equational theories, such as the theory of associative and commutative function symbols (AC), the equational parts of linear arithmetic (ArithS) over rationals and integers, and the equational part of enumerated data types (EnumS). A key feature of this combination approach is that, we only maintain a single union-find data structure for all the theories with green boxes. These theories have no internal state and they don't need to infer implied interface equalities, as in Nelson-Oppen's framework. Note that, the UF module also handles disequalities and partitions the terms of the input problem into equivalence classes. The Shostak-like approach is built upon two algorithms:

$$\mathsf{CC(X)} \quad + \quad \mathsf{AC(X)}$$

where,

- CC(X) is an extension of a congruence closure algorithm with a Shostak theory X [34]. This algorithm was formally proved in Coq [118]
- AC(X) is the combination framework presented in Chapter 4

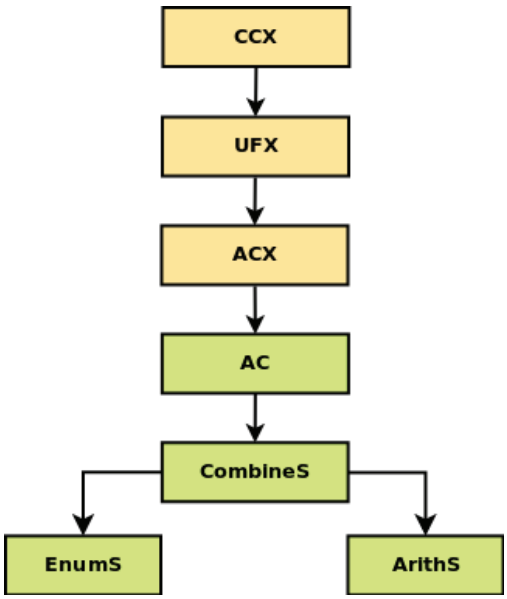

Figure 5.3: The architecture of the Shostak-like combination

The rest of this section is devoted to the description of these components. Note, however, that our presentation omits a lot technical details, for sake of the clarity.

Sorts representation. We assume given an OCaml type, called `sort`, used to represent sorts information. Its shape is given in Figure 5.4: a sort is either an integer `Sint`, a rational `Srat`, an abstract `Sabstract s`, an enumeration `Senum(s,l)`,

```
1 type sort =
2   | Sint
3   | Srat
4   | Sabstract of string
5   | Senum of string * string list
6   | Sarray of sort * sort
```

Figure 5.4: The minimal interface of sorts

or an array `Sarray`$(\mathtt{s}_1, \mathtt{s}_2)$. The list `l` of an enumeration contains the list of its constructors. The variables $\mathtt{s}_1$ and $\mathtt{s}_2$ of an array provide the sort information of indices and values, respectively.

Terms representation. We also assume given an OCAML type, called `term`, used to represent and manipulate terms. The shape of this type is shown in Figure 5.5. A term is always decorated with its sort (line 2). Its head symbol is either arithmetic (lines 4-6), an array access or update (lines 7-8), an AC function symbol (line 9), an enumeration (line 10) or uninterpreted (line 11). We assume given a comparison function `term_compare`, a set `SetT` and a map `MapT` over terms.

```
1  type term = { dsc : term_desc ; srt : sort }
2
3  and term_desc =
4    | Plus  of term * term
5    | Minus of term
6    | Number of number
7    | Get of term * term
8    | Set of term * term * term
9    | AcApp of string * term * term
10   | Enum of string
11   | App of string * term list
```

Figure 5.5: The minimal interface of terms

5.1.1 The interface of Shostak theories

The minimal interface we require for Shostak theories is given in Figure 5.6. The function `is_mine` tells whether the head symbol of the given term is owned by the theory. The functions `canon` and `solve` implement the canonizer and the solver routines. Finally, `replace` implements the replacement mechanism.

```
1  module type SHOSTAK_INTERFACE = sig
2    val is_mine : term -> bool
3    val canon   : term -> term
4    val solve   : term -> term -> (term * term) list
5    val replace : term -> term -> term -> term
6  end
```

Figure 5.6: The minimal interface required for Shostak theories

The module ArithS implements the canonizer routine described in Section 2.3.2. For the solver, it implements Gaussian elimination for rationals and Omega-Test's equalities solver for integers. The canonizer function for the module EnumS is the identity function. The solver follows the lines of the description given in Section 2.3.4. Finally, the module CombineS implements a global canonizer using the

individual theories' canonizers. The global solver is an *experimental algorithm*[1] that aims at combining solver routines for well-sorted Shostak theories.

5.1.2 The interface of the module AC

Figure 5.7 shows the simplified interface of AC. It is very similar to that of Shostak theories, except that it does not provide a solver routine, but only a wrapper as described in Section 4.2.3. The function `replace` implements the notion of *canonized rewriting* defined in Section 4.2.2.

```
module type AC_INTERFACE = sig
  val is_mine : term -> bool
  val canon   : term -> term
  val w_solve : term -> term -> (term * term) list
  val replace : term -> term -> term -> term
end
```

Figure 5.7: The simplified interface of AC

5.1.3 The interface and the implementation of ACX

Figure 5.8 shows the simplified interface of ACX. The environment is a dictionary from terms to terms representing a rewriting system. The actual implementation is inspired by AC(X). However, two major differences have to be pointed out:

- The environment contains only rules with AC-headed left-hand sides. In fact, the equivalence properties[2] of the free theory of equality are handled by the module UFX.
- The congruence axioms of the free theory of equality is delegated to CCX.

```
module type ACX_INTERFACE = sig
  type env
  val empty : env
  val normal_form : term -> env -> term
  val assume : term -> term -> env -> (term * term) list * env
end
```

Figure 5.8: The simplified interface of ACX

[1] This is still an ongoing work !
[2] *i.e.* reflexivity, symmetry and transitivity

The function `normal_form` computes canonical normal forms of terms *w.r.t* a given rewriting system. The function `assume` — sketched in Figure 5.9 — processes new equalities. It implements the following strategy of AC(X):

$$\textbf{Sim}^* \ (\textbf{Tri} \mid \textbf{Bot} \mid (\textbf{Ori} \ (\textbf{Com Col Ded})^*))$$

Given two terms s and t in canonical normal form, the equality $s \approx t$ is "solved" using the wrapper function provided by AC. If this equality is neither trivial nor inconsistent, a list of rewriting rules is returned. Following the strategy above, **Com**pose is applied to reduce right-hand sides of rules in the environment, and **Col**lapse is used to compute collapsing rules. The list $\texttt{l}_1$ contains the collapsed equalities that should be replayed. At line 5, the list of rules is partitioned as fol-

```
1 let rec assume s t env  =
2   let rules   = AC.w_solve s t in
3   let env     = compose env rules in
4   let env, l1 = collapse env rules in
5   let ac, emp = partition_rules rules in
6   let l2      = head_cp env ac in
7   let env     = append_rules env ac in
8   let emp2, env = replay_equalities env (l1 @ l2) in
9   emp @ emp2, env
```

Figure 5.9: The simplified implementation of `assume_eq`

lows: a rewriting rule $l \to r$ is put in the list `ac` (*resp.* `emp`) if and only if $\Lambda(l)$ is an AC (*resp.* uninterpreted) function symbol. The rules in `ac` are then used to compute head critical pairs and added to ACX's environment.

Deduced and collapsed equalities are recursively assumed at line 8. This yields a new term rewriting system and a list $\texttt{emp}_2$ of rules whose left-hand sides are uninterpreted-headed terms. The function `assume` finally returns the concatenation $\texttt{emp} \ @ \ \texttt{emp}_2$ and a new AC environment. Note that @ stands for the list concatenation operator.

5.1.4 The interface and the implementation of UFX

Union-find efficiently handles the reflexivity, the symmetry and the transitivity of the free theory of equality. UFX is an extension of a union-find with AC and a Shostak theory. This is achieved via the integration of the AC(X) procedure in a union-find data structure: instead of just merging the equivalence classes of equal terms as done in union-find, the UFX module uses the rewriting system provided

by ACX to construct equivalence classes modulo both the equivalence properties of the free theory of equality, the AC theory and the Shostak theory X. Note that UFX is extended to additionally handle disequalities.

Figure 5.10 shows the interface of the module UFX. The type `env` represents the environment of the data-structure. The value `empty` is the empty environment. UFX is incremental: the function `add_term` is used to "register" the terms of the problem in the environment on the fly.

```
1  module type UFX_INTERFACE = sig
2    type env
3    val empty : env
4
5    val add_term : term -> env -> env
6    val union : term -> term -> env -> env
7    val find  : term -> env -> term
8
9    val distinct  : term -> term -> env -> env
10   val are_equal : term -> term -> env -> bool
11   val are_distinct : term -> term -> env -> bool
12   val class_of : term -> env -> term list
13 end
```

Figure 5.10: The interface of the UF module

The functions `union` and `distinct` assume new equalities and disequalities, respectively. The function `find` returns the canonical normal forms of terms *w.r.t* a given environment. The function `class_of` returns the equivalence classes associated to terms. Finally, the functions `are_equal` and `are_distinct` test whether two terms are known to be equal or distinct in UFX, respectively. Note that, the terms manipulated by these functions are supposed to be previously registered using `add_term`.

The environment of UFX — shown in Figure 5.11 — is composed of four fields:

- `repr` is a map of terms that associates to each added term its current normal form, called its *representative*
- `ac` is the rewriting system maintained by ACX. As stated in the previous section, this field contains rules with AC-headed left-hand sides only.
- `dis` is a map of terms that associates to each representative the set of initial terms from which it is known to be distinct

- `cls` is a map that associates each representative to a list of initial terms that are equivalent modulo AC, X and the equivalence properties of the equality.

```
1  type env =
2    { repr : term MapT.t;       (* map: term -> term *)
3      ac   : ACX.env;            (* AC's environment *)
4      dis  : SetT.t MapT.t;     (* map: term -> Set(term) *)
5      cls  : term list MapT.t   (* map: term -> term list *) }
```

Figure 5.11: The environment of UFX

The implementation of `add_term` is shown in Figure 5.12. Given a term `t` not in UFX, the function first computes its representative *w.r.t.* the fields `repr` and `ac`, using the auxiliary function `normal_form`. The fields of the environment are then updated. Note that, the sub-terms of `t` are supposed to be previously added.

```
1  let add_term t env =
2    if MapT.mem t env.repr then env
3    else
4      let r = normalize t env.repr env.ac in
5      let dis =
6        if MapT.mem r env.dis then env.dis
7        else MapT.add r SetT.empty env.dis
8      in
9      let cls_r = try MapT.find r env.cls with Not_found -> [] in
10     {env with
11       repr = MapT.add t r env.repr;
12       cls  = MapT.add r (t::cls_r) env.cls;
13       dis  = dis}
```

Figure 5.12: Implementation of the function `add_term`

The implementation of the function `union` is shown in Figure 5.13. Given two terms `s` and `t`, this function starts by retrieving their respective representatives. If these representatives are not known to be distinct (line 15) or equal (line 18), the function `assume` of ACX is called on these representatives. If no theory inconsistency is encountered, this function returns a new rewriting system and a set of rewriting rules, where the left-hand sides are uninterpreted. The term rewriting system and the rewriting rules are then used to update the other fields of UFX's environment at line 11, thanks to function `update_env`.

Finally, the implementation of the functions `distinct` and `find` are shown in Figure 5.14. The first function starts by checking that the given terms have not the

```
let union s t env =
  let rs = find s env in
  let rt = find t env in
  if term_compare rs rt = 0 then env
  else
    let dis_of_s = MapT.find rs env.dis in
    let dis_of_t = MapT.find rt env.dis in
    let inters   = SetT.inter dis_of_s dis_of_t in
    if SetT.is_empty inters then
      let rules, ac = ACX.assume rs rt env.ac in
      update_env {env with ac = ac} rules
    else
      raise Bottom (* s and t are known to be distinct *)
```

Figure 5.13: Implementation of the function `union`

same representative. Then, it updates the field `dis` thanks to the auxiliary function `add_to_distinct_of`. The function `find` simply canonizes the given term and looks for its representative in `repr`. Note that, the function MapT.`find` raises the exception `Not_found` if the term was not already added to UFX's environment.

```
let find t env = MapT.find t env.repr
```

```
let distinct s t env =
  let rs = find s env in
  let rt = find t env in
  if term_compare rs rt = 0 then raise Bottom
  else
    let dis1 = add_to_distinct_of rs t env.dis in
    let dis2 = add_to_distinct_of rt s dis1 in
    {env with dis = dis2}
```

Figure 5.14: Implementation of the function `distinct` and `find`

5.1.5 The interface and the implementation of CCX

Congruence closure algorithms are very efficient for reasoning modulo the free theory of equality. The CCX module follows the lines of CC(X). This framework extends a congruence closure algorithm with a Shostak theory X. The extension rests on the integration of the Shostak theory X in the union-find data structure, used at the core of congruence closure algorithms. The detailed formalization of CC(X) is given in [85], page 62. We recall here the main ideas.

The internal representation of CCX's environment is shown in Figure 5.15. The first field is the environment of UFX. The second one is a map from terms to set of terms. An entry $t \mapsto \{\ldots, f_i(t_1, \ldots, t_n), \ldots\}$ of `used_by` means that: the term t is a maximal alien of every representative of t_j *w.r.t.* the field `uf`, where $j \in [|1, n|]$.

```
type env =
  { uf      : UFX.env;
    used_by : SetT.t MapT.t }
```

Figure 5.15: The environment of CCX

The interface of CCX is shown in Figure 5.16. This module is incremental: the function `add_term` is used to register the terms of the problem on the fly. It may deduce equalities by congruence, which are immediately assumed. The functions `assume_eq` and `assume_dis` allow to assume equalities and disequalities, respec-

```
module type CCX_INTERFACE = sig
  type env
  val empty      : env
  val add_term   : term -> env -> env
  val assume_eq  : term -> term -> env -> env
  val assume_dis : term -> term -> env -> env
  val ufx_of     : env -> UFX.env
end
```

Figure 5.16: The simplified interface of CC

tively. Again, the function `assume_eq` may deduce equalities by congruence, which are recursively assumed. The example below illustrates a run of CC(X):

Example 63. *Let us show that the following formula is unsatisfiable:*

$$a \approx b \quad \wedge \quad f(a,c) \approx 1 \quad \wedge \quad f(b,c) \approx 2$$

where a, b, c are constants of sort `int`*, and f is a function symbol of sort* $\texttt{int} \times \texttt{int} \to \texttt{int}$.

Scenario a. *Suppose that equalities are treated from left to right. When the term $f(b,c)$ is added, UFX's environment contains the following information:*

$$\begin{array}{lcl} \texttt{uf.repr} & : & a \mapsto b, \quad b \mapsto b, \quad f(a,c) \mapsto 1, \quad f(b,c) \mapsto f(b,c) \\ \texttt{used\_by} & : & a \mapsto \emptyset, \quad b \mapsto \{f(a,c), f(b,c)\}, \quad c \mapsto \{f(a,c), f(b,c)\} \end{array}$$

Since the terms b and c are "used by" both $f(a,c)$ and $f(b,c)$, the function `add_term` *tests whether $f(a,c) \approx f(b,c)$ follows by congruence. This is actually the case, since $a \approx b$. The inconsistency $(1 \approx 2)$ is then derived.*

Scenario b. *Suppose that equalities are treated from right to left. Before the equality $a \approx b$ is assumed,* UFX*'s environment contains the following information:*

$$\begin{array}{lcl} \texttt{uf.repr} & : & a \mapsto a, \quad b \mapsto b, \quad c \mapsto c, \quad f(a,c) \mapsto 1, \quad f(b,c) \mapsto 2 \\ \texttt{used\_by} & : & a \mapsto \{f(a,c)\}, \quad b \mapsto \{f(b,c)\}, \quad c \mapsto \{f(a,c), f(b,c)\} \end{array}$$

At this point, it is not necessary to check whether $f(a,c) \approx f(b,c)$, because a and b are not "used by" $f(b,c)$ and $f(a,c)$, respectively. However, when $a \approx b$ is treated, the new environment contains the following information:

$$\begin{array}{lcl} \texttt{uf.repr} & : & a \mapsto b, \quad b \mapsto b, \quad c \mapsto c, \quad f(a,c) \mapsto 1, \quad f(b,c) \mapsto 2 \\ \texttt{used\_by} & : & a \mapsto \emptyset, \quad b \mapsto \{f(b,c), f(a,c)\}, \quad c \mapsto \{f(a,c), f(b,c)\} \end{array}$$

Consequently, the function `assume_eq` *deduces that $f(a,c) \approx f(b,c)$ by congruence, which allows us to conclude.*

5.2 The Nelson-Oppen-like combination framework

When a theory cannot be integrated in the Shostak-like method, we use another approach reminiscent of Nelson-Oppen's framework. This approach allows us, for instance, to combine the theory of arrays (ArrNS), the inequalities of linear arithmetic (ArithNS) and the non convex part of enumerated data types (EnumNS).

Figure 5.17 shows the architecture of this combination framework. Contrarily to the first technique, each single procedure maintains an internal environment. We present in the rest of this section the interface we require for these theories and sketch the implementation of some interesting functions for each module.

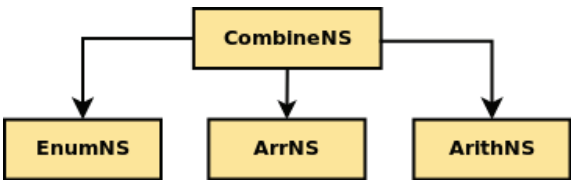

Figure 5.17: The architecture of the Nelson-Oppen-like combination

Literals representation. In the rest of this chapter, literals will be represented using the OCAML types given in Figure 5.18. A literal is composed of two terms and a kind. The kind indicates whether a literal is an equality `Eq`, a disequality `Dis`, a large inequality `Le` or a strict inequality `Lt`. We assume given a comparison function `literal_compare` over literals and a function `literal_neg` that returns negations of literals. We also assume given a map, called `MapL`, over literals.

```
type literal_kind = Eq | Dis | Le | Lt
type literal = literal_kind * term * term
```

Figure 5.18: The minimal interface of literals

5.2.1 The interface of Nelson-Oppen like theories

Figure 5.19 shows the interface we require for the theories we combine using the Nelson-Oppen like approach. As usual, the function `add_term` inserts the terms of the problem in theories' states. The function `assume` is used to assume literals. Note that, this function takes UFX's environment as argument and may return a list of implied literals. The function `case_split` is used to ask the theory for a case-split over finite domains. It returns a possible choice if it exists, or `None` otherwise.

```
module type NS_INTERFACE = sig
  type env
  val empty      : env
  val add_term   : term -> env -> env
  val assume     : literal -> UFX.env -> env -> literal list * env
  val case_split : env -> literal option
end
```

Figure 5.19: The required interface for non convex/non equational theories

We call the theories having this interface *Nelson-Oppen like theories*, because they have to deduce all implied (disjunction) of literals. However, contrary to Nelson-Oppen's framework, we are not only interested in implied literals over shared variables, as equality reasoning over convex theories is entirely delegated to the Shostak-like combination approach.

5.2.2 The non convex part of enumerated data types

The implementation of the main functions in EnumNS are given in Figures 5.20, 5.21 and 5.22. The environment is a dictionary that associates to each term t of sort $\texttt{Senum}(\texttt{s}, \texttt{constrs})$, a sub-list of `constrs` that represent the possible values for t. The function `add_term` simply fills the map with new entries.

```
type env = (string list) MapT.t

let add_term t env =
  match t.srt with
  | Senum (s,constrs) ->
    if MapT.mem t env then env else MapT.add t constrs env
  | _ -> env
```

Figure 5.20: The implementation of the function EnumNS.`add_term`

The function `assume`, sketched in Figure 5.21, uses two auxiliary functions that are not hard to implement. For instance, `assume_eq` acts as follows when applied on two terms t_1 and t_2 that are not owned by the theory of enumerated data types:

1. it retrieves the sets st_1 and st_2 of values associated to t_1 and t_2 respectively
2. it computes the intersection $st = st_1 \cap st_2$
3. if st is empty, it raises the exception `Bottom`
4. otherwise, it updates the entries of t_1 and t_2 with st
5. if $st = \{c\}$ is a singleton, it infers an entailed equality $t_1 \approx c$.

```
let assume (lit, s, t) uf env =
  if has_enum_sort s then
    match lit with
    | Eq  -> assume_eq  (UFX.find s uf) (UFX.find t uf) env
    | Dis -> assume_dis (UFX.find s uf) (UFX.find t uf) env
    | _ -> [], env
  else [], env
```

Figure 5.21: The implementation of the function EnumNS.`assume`

The function `case_split`, given in Figure 5.22, looks for a binding of the form $s \mapsto \{c_1,\ c_2,\ \cdots\}$ in `env`. If such a binding exists, it suggests a case-split $s \approx c_i$. Note that, the cases $s \mapsto \emptyset$ and $s \mapsto \{c\}$ are handled by the function `assume`.

```
exception Found of term * string

let case_split env =
  try
    MapT.iter
      (fun s constrs ->
        match constrs with
        | [] | [_]    -> assert false
        | cons::_::_ -> raise (Found (s,cons))
      ) env;
    None
  with Found (s,cons) ->
    Some (Eq, s, {s with dsc = App (cons,[])})
```

Figure 5.22: The implementation of the function EnumNS.`case_split`

5.2.3 The theory of functional arrays

The decision procedure for the theory of functional arrays works by instantiation of its set of axioms. It generates additional ARR-valid ground instances using:

- the axioms of the theory of extensional functional arrays (ARR) introduced in Section 2.3.3
- the ground terms appearing in the problem

The quantified variables of these axioms are not instantiated for every possible well sorted terms, because a too permissive instantiation would saturates the memory. One should be as restrictive as possible, when instantiating these axioms, while preserving completeness. In practice, ground instances are computed using matching algorithms modulo equality. The restriction of the instantiation is achieved using the mechanism of *patterns* (*a.k.a.* triggers). In other words, each axiom of the set ARR_{NS} given in Section 2.3.3 is instantiated as follows[64, 54]:

- Axiom 1 is instantiated with a substitution $\sigma = \{x_a \mapsto a,\ x_v \mapsto v,\ x_i \mapsto i\}$ if there exists a ground term $set(a, v, i)$ in the problem's context
- Axiom 2 is instantiated with $\sigma = \{x_a \mapsto a,\ x_v \mapsto v,\ x_i \mapsto i,\ x_j \mapsto j\}$ if there exists a ground term $get(set(a, c, i), j)$ in the problem's context
- Axiom 2 is instantiated with $\sigma = \{x_a \mapsto a,\ x_v \mapsto v,\ x_i \mapsto i,\ x_j \mapsto j\}$ *modulo equality* if there exists two ground terms $get(a', j)$ and $set(a, v, i)$ such that a' and $set(a, v, i)$ (*resp.* and a) are in the same equivalence class.

- Axiom 3 is instantiated with a substitution $\{x_a \mapsto a,\ x_b \mapsto b\}$ for every couple of arrays a and b that are known to be distinct.

The environment of the module ArrNS and the implementation of the functions `case_split` and `add_term` are shown in Figure 5.23. The environment is a record. The fields *gets* and *sets* are sets of terms. The field *instances* is a map from literals to lists of literals. A binding $a \mapsto [a_1;\ a_2;\ \ldots;\ a_n]$ in this map represents the implication $a \Rightarrow (a_1 \wedge a_2 \wedge \ldots \wedge a_n)$. The literal a is called the *premise* of the binding and the a_i are its *conclusions*. The function `case_split` simply returns a

```
1  type env =
2    { gets : SetT.t; sets : SetT.t; instances : literal list MapL.t }
3
4  let case_split env =
5    if MapL.is_empty env.instances then None
6    else let prem, concl = MapL.choose env.instances in Some prem
7
8  let add_term t env =
9    match t.dsc with
10   | Get _ -> {env with gets = SetT.add t env.gets}
11   | Set _ -> {env with sets = SetT.add t env.sets}
12   | _ -> env
```

Figure 5.23: The state and the functions `case_split` and `add_term` of ArrNS

premise from the map *instances*., while the function `add_term` fills the components *gets* and *sets* with adequate terms.

The function `assume` handles the instantiation of the axioms in ARR_{NS} (lines 8, 9, 10) as explained above. It also returns the list of literals that are consequences of the assumed fact, thanks to the auxiliary function `conclusions_of_premise`.

```
1  let conclusions_of_premise env a =
2    try
3      let implied = MapL.find a env.instances in
4      implied, {env with instances = MapL.remove a env.instances}
5    with Not_found -> [], env
6
7  let assume a uf env =
8    let env = instanciate_axiom_1 env uf a in
9    let env = instanciate_axiom_2 env uf a in
10   let env = instanciate_axiom_3 env uf a in
11   conclusions_of_premise env a
```

Figure 5.24: The implementation of the function ArrNS.`assume`

5.2.4 Inequalities of linear integer arithmetic

The environment of the module ArithNS is shown in Figure 5.25. It consists of two maps from terms to intervals. The first one is used to integers and the second one for rationals.

```
type env = { ints: Interval.t MapT.t; rats: Interval.t MapT.t }
```

Figure 5.25: The environment of the module ArithNS

The minimal interface of the module Interval used in the environment of ArithNS is given in Figure 5.26. The function `size` returns the size of the given interval and the function `min` returns its minimal value.

```
module type INTERVAL_INTERFACE = sig
  type t
  val size : t -> int (* returns a negative integer if t is unbounded *)
  val min  : t -> number
end
```

Figure 5.26: The minimal interface of Interval

The implementation of the function `case_split`, given in Figure 5.27, is similar to that of EnumNS. When some interval in the field $ints$ is bounded and has at least two possible values, a case-split analysis is suggested by the function.

```
exception Found of term * number

let case_split env =
  try
    MapT.iter
      (fun s interv ->
        if Interval.size interv > 1 then
          raise (Found (s, Interval.min interv))
      ) env.ints;
    None
  with Found (s,num) ->
    Some (Eq, s, {s with dsc=Number num})
```

Figure 5.27: The implementation of the function ArithNS.`case_split`

The implementation of the function `assume` is sketched in Figure 5.28. It uses two auxiliary functions to handle integers and rationals respectively. The function

`assume_int` first updates the map $ints$ by taking into account the assumed literal a. Then, it attempts to infer better bounds for the terms used as keys of the map $ints$. At this point, we can either used the Fourier-Motzkin algorithm or the simplex-based approach described in Chapter 3. The equalities that may be implied at lines 2 and 3 are extracted at line 4. The function `add_term` is not used by ArithNS. It returns the given environment without modifying it.

```
1  let assume_int a uf env =
2    let ints = add_to_env a uf env.ints in
3    let ints = tighten_bounds ints in
4    let eqs  = extract_inferred_equalities ints in
5    eqs, {env with ints = ints}
6
7  let assume a uf env =
8    if has_int_sort a then assume_int a uf env
9    else if has_rat_sort a then assume_rat a uf env
10   else [], env
```

Figure 5.28: The implementation of the function ArithNS.`assume`

5.2.5 Implementation of the module CombineNS

Figure 5.29 shows the implementation of some functions in CombineNS. Contrary to CombineS, this module does not perform a subtle combination of the individual theories. The function `case_split` simply asks each theory for a case-split. The function `assume` (*resp.* `add_term`) propagates assumed literals (*resp.* added terms) to individual theories.

```
1  type env = EnumNS.env * ArraysNS.env * ArithNS.env
2
3  let case_split (e1, e2, e3) =
4    match EnumNS.case_split e1 with
5    | (Some _) as cs -> cs
6    | None ->
7      match ArraysNS.case_split e2 with
8      | (Some _) as cs -> cs
9      | None -> ArithNS.case_split e3
10
11 let assume a uf (e1, e2, e3) =
12   let l1, e1 = EnumNS.assume   a uf e1 in
13   let l2, e2 = ArraysNS.assume a uf e2 in
14   let l3, e3 = ArithNS.assume  a uf e3 in
15   l1 @ l2 @ l3 , (e1, e2, e3)
```

Figure 5.29: The implementation of the dispatcher CombineNS

5.3 The Combinator module

Now, it remains to describe the implementation of the Combinator. This module is made of two sub-components, as shown in Figure 5.30. The Combine module deals with information exchange between the two main combination approaches of ALT-ERGO. The CSA module implements the case-split analysis mechanism for non-convex theories.

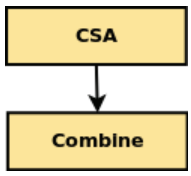

Figure 5.30: The architecture of the Combinator module

5.3.1 Implementation of the module Combine

The simplified interface of Combine is given in Figure 5.31. It is composed of value `empty` which represents the empty environment, a function `assume` for assuming literals, and a function `case_split` that asks non-convex theories for case-split analysis.

```
1 module type COMBINE_INTERFACE = sig
2   type env
3   val empty : env
4   val assume : env -> literal -> env
5   val case_split : env -> literal option
6 end
```

Figure 5.31: The interface of the module Combine

Figure 5.32 shows the implementation of the main functions of this module. The environment is a pair containing the state of CCX and the state of CombineNS. Given a literal a, the function `assume` starts by recursively adding the terms and the sub-terms of a in the environment. Then, a is passed to CCX.`assume` if this literal is an equality or a disequality. After that, CombineNS propagates a to its individual modules with the current state of UFX. The literals that CombineNS may infer are recursively assumed.

```
type env  = { cc : CCX.env; ns : CombineNS.env }

let case_split env = CombineNS.case_split env.ns

let rec assume env a =
  let cc, ns = add_terms_rec a env in
  let cc = match a with
    | (Eq,  s,t) -> CCX.assume_eq  s t cc
    | (Dis, s,t) -> CCX.assume_dis s t cc
    | _                -> cc
  in
  let uf = CCX.ufx_of cc in
  let implied, ns = CombineNS.assume a uf ns in
  let env = {cc=cc; ns=ns} in
  List.fold_left assume env implied
```

Figure 5.32: The implementation of some functions in Combine

5.3.2 Implementation of the module CSA

In ALT-ERGO, the case-split analysis is not delegated to the SAT solver engine. This task is performed by the CSA module, situated between the SAT and the background decision procedures. The simplified interface of this module is shown in Figure 5.33.

```
module type CSA_INTERFACE = sig
  type env
  val empty  : env
  val assume : env -> literal -> env
end
```

Figure 5.33: The interface of the module CSA

A basic implementation of this interface is given in Figure 5.34. The environment contains two fields of type Combine.env. The `real` field represents the actual state of the decision procedure. The `spec` field is a *speculative* state. It is equal to the `real` state, plus some case-split assumptions made during the search. Of course, the SAT solver engine may contradict one of these assumptions during its search for a model.

```
1  type env =  { real : Combine.env; spec : Combine.env }
2
3  let rec cs_analysis spec =
4    match Combine.case_split spec with
5    | None   -> spec
6    | Some a ->
7      try cs_analysis (Combine.assume spec a)
8      with Bottom -> cs_analysis (Combine.assume spec (literal_neg a))
9
10 let assume_spec real spec a =
11   let spec =
12     try Combine.assume spec a
13     with Bottom -> real
14   in
15   {real = real; spec = cs_analysis spec}
16
17 let assume env a =
18   let real = Combine.assume env.real a in
19   assume_spec real env.spec a
```

Figure 5.34: The implementation of some functions in CSA

When a literal a is assigned to $true$ by the SAT solver, it is added to both `real` and `spec` (lines 18 and 19). If the speculative state becomes inconsistent while assuming a, it is reset to `real` (line 13). After that, the case-split analysis is started in order to find a satisfying assignment for the values that are in finite domains. The implementation given below is very naive because we may redo the same case-splits at line 15 each time we reset `spec` to `real` at line 13. A cleverer implementation can easily avoid this issue. Another source of inefficiency of the presentation below lies in the function `cs_analysis` that explores all the branches when searching for a model. The actual implementation of this function in ALT-ERGO is more elaborated. In fact, it uses conflict analysis and non-chronological backtracking to prune the search space. Note that we can safely reuse the state `spec` at line 8 when searching for a model in negative branches, because the data structures we used in the internal modules are persistent. Consequently, we are sure they are not modified by side-effects at line 7. Note that, we first perform case-split analysis on values that are in smaller domains, in practice.

Our case-split analysis mechanism is completely hidden for the SAT solver. When there in no possible case-split assignment for the `real` field, an uncaught exception `Bottom` is raised by `cs_analysis` at line 8. Seeing that the background decision procedure disagrees with its current boolean model, the SAT solver module repairs it and prunes the search space, before continuing its exploration.

5.4 Evaluation

In order to measure the improvements made in ALT-ERGO during this thesis[3], we benchmark the current release of ALT-ERGO and compare its performances with:

- older releases of the prover, starting from version 0.9 published in July 2009
- some state-of-the-art SMT solvers with quantifiers support, such as CVC3 (v 2.4.1), YICES (v 1.0.38) and Z3 (v 3.2 and v 4.2)

For our experiments, we used logical formulas generated from WHY3's gallery of programs [4]. This gallery contains 82 programs, from which WHY3 generated 1920 verification conditions. Almost all of them are valid. They are generated in

- the native polymorphic input language of ALT-ERGO
- the native input language of CVC3
- the native input language of YICES
- the SMT-LIB2 input language for Z3

The measures were obtained on a 64-bit machine with a quad-core Intel Xeon processor at 3.2 GHz and 24 GB of memory. Provers were given a time limit of 30 seconds and a memory limit of 2 GB for each test.

Experiments' results. The results of our experiments are reported in Figure 5.35 and Figure 5.36. The first column shows the provers we used with their versions. In columns 2 and 3, we report the number of verification conditions that are proved valid by each prover and the corresponding accumulated time. Columns 4 and 5 report the numbers of VCs for which the provers returned "unknown" and the accumulated time, respectively. In column 6 (*resp* 7), we report the number of VCs on which the provers timeout-ed (*resp.* encountered an error during their execution, such as "out of memory").

Results analysis. Figure 5.35 shows that ALT-ERGO has made a significant progress since version 0.9. In particular, this observation holds for both the total number of proved VCs, and the number of proved VCs per second. We also notice on Figure 5.36 that ALT-ERGO is performing better than other provers on our benchmark.

[3] Note that some enhancements have been made by other people

[4] http://toccata.lri.fr/gallery/why3.en.html

	v 0.95.1	v 0.94	v 0.93	v 0.92.2	v 0.91	v 0.9
valid	**1841**	1811	1773	1737	1685	1306
time	**362**	465	411	527	555	290
unknown	**20**	25	24	22	21	233
time	**13**	31	25	33	45	59
timeout	**59**	83	99	128	175	328
errors	**0**	1	24	33	39	53

Figure 5.35: The comparison of different releases of ALT-ERGO using formulas generated from WHY3's gallery of programs.

	ALT-ERGO **v 0.95.1**	CVC3 v 2.4.1	YICES v 1.0.38	Z3 v 3.2	Z3 v 4.2
valid	**1841**	1767	515	1592	1524
time	**362**	357	103	446	256
unknown	**20**	30	1029	3	1
time	**13**	134	1247	0.2	0.1
timeout	**59**	107	164	295	305
errors	**0**	16	212	30	90

Figure 5.36: The comparison of ALT-ERGO, Z3, CVC3 and YICES using formulas generated from WHY3's gallery of programs.

CHAPTER 6

Conclusion

An alternative simplex-based procedure for deciding QF-LIA

In Chapter 3, we have presented a novel algorithm for deciding the theory of quantifier-free linear integer arithmetic. Our procedure is composed of two parts. The master part handles equalities solving, the maintenance and the inference of integers bounds for inequalities' affine forms.

The inference of precise bounds for the affine forms is achieved by computing particular constant positive linear combination of them. These combinations are encoded as rational optimization problems that simulate particular runs of Fourier-Motzkin's algorithm. The resolution of these problems is delegated to a simplex-based oracle.

When some inequalities's affine forms are bounded, a case-split analysis is used to check whether there exists an integer solution in these bounds for the initial conjunction of literals. If none of the affine forms can be bounded, we deduce that the given constraints admits a solution in the integers. To summarize, the main scientific contributions of this work are:

1. A theorem stating that: if there is no constant positive linear combination of a conjunction of constraints' affine forms, then the convex polytope defined by these constraints contains infinitly many integer points. As far as we know, this theorem is not in the literature. Moreover, we can exhibit an integer model in this case, as its proof is constructive.

2. An efficient simplex-based algorithm for computing constant positive linear combinations of affine forms. This algorithm was initially inspired by the Fourier-Motzkin method. However, it is more efficient and scales better in practice. In addition, although it uses a rational simplex, our procedure is not really a simplex extension, like branch-and-bound or cutting-planes.

Our framework is easily extensible with intervals calculus, since it is based on bounds maintenance. This particularity allows us to additionally incorporate non-linear arithmetic reasoning in practice.

We believe that additional improvements can be made in our procedure. In fact, the actual oracle in neither incremental nor backtrackable, which can be annoying in an SMT solver. In addition, our case-split analysis technique does not behave well on very large bounded intervals. In the near future, we plan to to address all these issues.

We think that our procedure is complementary with those extending a rational simplex to decide linear integer arithmetic, such as branch-and-bound, cutting-planes. We believe that the combination of these mature techniques with ours would be an interesting idea to investigate.

Integrating rewriting techniques in SMT

In Chapter 4, we have presented a combination framework, called AC(X), that enables reasoning in the union of the free theory of equality with uninterpreted symbols, the AC theory and a Shostak theory X that fulfills reasonable ordering constraints. Our framework consists of a modular extension of ground AC completion with the theory X. The key ideas of this extension are:

1. the use of the solver routine provided by the theory X to handle equalities instead of just orienting them
2. the extension of rewriting modulo AC, used by ground AC completion, with the canonizer function of X.

AC(X) can be seen as an adaptation of normalized completion: an extension of AC completion with a first-order convergent rewriting system. However, our framework is much more simpler and has the nice termination property, while normalized completion may not terminate.

In order to fully integrate AC(X) in the ALT-ERGO SMT solver, we have shown that the prover's axioms instantiation mechanism — based on E-matching — has to be extended modulo AC. Using a simple example, we have shown that this extension is not immediate, because E-AC-matching is not finitary. We plan to investigate incremental instantiation alternatives to get rid of this issue.

In order to integrate additional rewriting techniques in SMT solvers, we have investigated the reuse of our combination technique to extend normalized completion with Shostak theories. This would allow us to handle user-defined first-order convergent rewriting systems. Unfortunately, we were quickly faced to several hard issues, such as the combination of unification algorithms for AC and Shostak theories. In the future, we plan to explore whether some reasonable restrictions

on the rewriting system or on some interesting Shostak theories would allow us to circumvent these issues.

Implementation

Satisfiability Modulo theories is a promising research topic. SMT solvers are now used in various domains, such as software and hardware verification, SMT-based model checking, test-case generation and compiler optimization.

In this thesis, we have considered the enhancement of our ALT-ERGO SMT solver to make it usable in the context of software verification. In Chapter 5, we have described the core decision procedures of our solver. In addition to FM-simplex and AC(X), we have implemented two decision procedures for the theories of enumerated data types and functional arrays, respectively.

The design of an efficient union-find data structure modulo theories — used at the heart of our Shostak-like combination framework — would significantly improve ALT-ERGO. We plan to explore this direction in the near future, as well as the combination of solvers routines for certain classes of Shostak theories.

In the future, we also plan to integrate additional interesting theories in the core of ALT-ERGO. For instance, these include the theory of recursive data types, the theory of floating-point numbers [26, 36, 69] and set theory.

Bibliography

[1] *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings*. IEEE Computer Society, 2001. (Cited on pages 162 and 163.)

[2] T. Achterberg, T. Koch, and A. Martin. MIPLIB 2003. *Operations Research Letters*, 34(4):361–372, 2006. (Cited on page 72.)

[3] S. B. Akers. Binary decision diagrams. *IEEE Trans. Comput.*, 27(6):509–516, June 1978. (Cited on page 3.)

[4] Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Kornilowicz, and Roberto Sebastiani. A sat based approach for solving formulas over boolean and linear mathematical propositions. In Andrei Voronkov, editor, *CADE*, volume 2392 of *Lecture Notes in Computer Science*, pages 195–210. Springer, 2002. (Cited on page 7.)

[5] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998. (Cited on pages 78 and 79.)

[6] L. Bachmair, A. Tiwari, and L. Vigneron. Abstract congruence closure. *Journal of Automated Reasoning*, 31(2):129–168, 2003. (Cited on pages 4, 77, 105 and 122.)

[7] Leo Bachmair, Nachum Dershowitz, and Jieh Hsiang. Orderings for equational proofs. In *Proc. 1st IEEE Symp. Logic in Computer Science, Cambridge, Mass.*, pages 346–357, June 1986. (Cited on page 90.)

[8] Leo Bachmair and Ashish Tiwari. Abstract congruence closure and specializations. In David A. McAllester, editor, *CADE*, volume 1831 of *Lecture Notes in Computer Science*, pages 64–78. Springer, 2000. (Cited on page 23.)

[9] Thomas Ball, Ella Bounimova, Rahul Kumar, and Vladimir Levin. Slam2: Static driver verification with under 4% false alarms. In Roderick Bloem and Natasha Sharygina, editors, *FMCAD*, pages 35–42. IEEE, 2010. (Cited on page 1.)

[10] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, MA, USA, 1988. (Cited on page 53.)

[11] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. (Cited on pages 1 and 10.)

[12] Mike Barnett, Robert DeLine, Bart Jacobs, Bor-Yuh Evan Chang, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *FMCO*, volume 4111 of *LNCS*, 2005. (Cited on page 1.)

[13] Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In Rajeev Alur and Doron A. Peled, editors, *Proceedings of the* 16^{th} *International Conference on Computer Aided Verification (CAV '04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518. Springer-Verlag, July 2004. Boston, Massachusetts. (Cited on page 7.)

[14] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag. (Cited on pages 5, 7, 8 and 33.)

[15] Clark Barrett, Morgan Deters, Leonardo de Moura, Albert Oliveras, and Aaron Stump. 6 years of SMT-COMP. *Journal of Automated Reasoning*, 2012. 10.1007/s10817-012-9246-5. (Cited on page 7.)

[16] Clark Barrett, David Dill, and Jeremy Levitt. Validity checking for combinations of theories with equality. pages 187–201. Springer-Verlag, 1996. (Cited on pages 6 and 7.)

[17] Clark Barrett, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Splitting on demand in SAT modulo theories. In Miki Hermann and Andrei Voronkov, editors, *Proceedings of the* 13^{th} *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR '06)*, volume 4246 of *Lecture Notes in Computer Science*, pages 512–526. Springer-Verlag, November 2006. Phnom Penh, Cambodia. (Cited on page 51.)

[18] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, 2010. (Cited on pages 9, 63 and 64.)

[19] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302, Berlin, Germany, July 2007. Springer. (Cited on pages 5, 7, 8 and 33.)

[20] Clark W. Barrett. *Checking Validity of Quantifier-Free Formulas in Combinations of First-Order Theories*. PhD thesis, Stanford University, January 2003. Stanford, California. (Cited on page 7.)

[21] François Bobot, Sylvain Conchon, Évelyne Contejean, Mohamed Iguernelala, Stéphane Lescuyer, and Alain Mebsout. The Alt-Ergo automated theorem prover, 2008. `http://alt-ergo.lri.fr/`. (Cited on pages 6, 7, 8, 10 and 33.)

[22] François Bobot, Sylvain Conchon, Evelyne Contejean, Mohamed Iguernelala, Assia Mahboubi, Alain Mebsout, and Guillaume Melquiond. A Simplex-based extension of Fourier-Motzkin for solving linear integer arithmetic. In Bernhard Gramlich, Dale Miller, and Ulrike Sattler, editors, *IJCAR 2012: Proceedings of the 6th International Joint Conference on Automated Reasoning*, volume 7364 of *Lecture Notes in Computer Science*, pages 67–81, Manchester, UK, June 2012. Springer. (Cited on page 34.)

[23] François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. *The Why3 platform*. LRI, CNRS & Univ. Paris-Sud & INRIA Saclay, version 0.64 edition, February 2011. `http://why3.lri.fr/`. (Cited on page 1.)

[24] Miquel Bofill, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. A write-based solver for sat modulo the theory of arrays. In Alessandro Cimatti and Robert B. Jones, editors, *FMCAD*, pages 1–8. IEEE, 2008. (Cited on page 27.)

[25] Marco Bozzano, Roberto Bruttomesso, Ro Cimatti, Tommi Junttila, Peter Van Rossum, Stephan Schulz, and Roberto Sebastiani. The mathsat 3 system. In *Automated Deduction: Proceedings of the 20th International Conference, volume 3632 of Lecture Notes in Computer Science*, pages 315–321. Springer, 2005. (Cited on page 7.)

[26] Martin Brain, Vijay D'Silva, Alberto Griggio, Leopold Haller, and Daniel Kroening. Interpolation-based verification of floating-point programs with

abstract CDCL. In *Static Analysis Symposium (SAS)*, volume 7935 of *LNCS*, pages 412–432. Springer, 2013. (Cited on page 149.)

[27] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzen, Alberto Griggio, and Roberto Sebastiani. Delayed theory combination vs. nelson-oppen for satisfiability modulo theories: a comparative analysis. *Annals of Mathematics and Artificial Intelligence*, 55(1-2):63–99, February 2009. (Cited on page 5.)

[28] Roberto Bruttomesso, Ro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The mathsat 4 smt solver (tool paper), 2008. (Cited on page 7.)

[29] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In Nir Piterman and Scott Smolka, editors, *Proceedings of TACAS*, volume 7795 of *LNCS*. Springer, 2013. (Cited on page 7.)

[30] Sylvain Conchon and Évelyne Contejean. The Alt-Ergo automatic theorem prover. `http://alt-ergo.lri.fr/`, 2008. APP deposit under the number IDDN FR 001 110026 000 S P 2010 000 1000. (Cited on page 108.)

[31] Sylvain Conchon, Évelyne Contejean, and Mohamed Iguernelala. Ground Associative and Commutative Completion Modulo Shostak Theories. In Andrei Voronkov, editor, *LPAR, 17th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, EasyChair Proceedings, Yogyakarta, Indonesia, October 2010. (short paper). (Cited on page 78.)

[32] Sylvain Conchon, Évelyne Contejean, and Mohamed Iguernelala. Canonized Rewriting and Ground AC Completion Modulo Shostak Theories. In Parosh A. Abdulla and K. Rustan M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 of *Lecture Notes in Computer Science*, pages 45–59, Saarbrücken, Germany, April 2011. Springer. (Cited on page 78.)

[33] Sylvain Conchon, Évelyne Contejean, and Mohamed Iguernelala. Canonized rewriting and ground AC completion modulo Shostak theories : Design and implementation. *Logical Methods in Computer Science*, 8(3):1–29, September 2012. Selected Papers of the Conference *Tools and Algorithms for the Construction and Analysis of Systems* (TACAS 2011), Saarbrücken, Germany, 2011. (Cited on page 78.)

[34] Sylvain Conchon, Évelyne Contejean, Johannes Kanig, and Stéphane Lescuyer. CC(X): Semantical combination of congruence closure with solvable theories. In *Post-proceedings of the 5th International Workshop on Satisfiability Modulo Theories (SMT 2007)*, volume 198(2) of *Electronic Notes in Computer Science*, pages 51–69. Elsevier Science Publishers, 2008. (Cited on page 127.)

[35] Sylvain Conchon and Sava Krstić. Strategies for combining decision procedures. *Theoretical Computer Science*, 354(2):187–210, 2006. Special Issue of TCS dedicated to a refereed selection of papers presented at TACAS'03. (Cited on page 6.)

[36] Sylvain Conchon, Guillaume Melquiond, Cody Roux, and Mohamed Iguernelala. Built-in treatment of an axiomatic floating-point theory for SMT solvers. In Fontaine and Goel [62], pages 12–21. (Cited on page 149.)

[37] Évelyne Contejean. A certified AC matching algorithm. In Vincent van Oostrom, editor, *15th International Conference on Rewriting Techniques and Applications*, volume 3091 of *Lecture Notes in Computer Science*, pages 70–84, Aachen, Germany, June 2004. Springer. (Cited on page 30.)

[38] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001. (Cited on page 38.)

[39] G. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963. (Cited on page 4.)

[40] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962. (Cited on pages 3 and 7.)

[41] L. de Moura and B. Dutertre. Yices: An SMT Solver. `http://yices.csl.sri.com`. (Cited on page 72.)

[42] Leonardo de Moura and Nikolaj Bjørner. Z3, an efficient SMT solver. `http://research.microsoft.com/projects/z3/`. (Cited on pages 8, 33 and 72.)

[43] Leonardo de Moura and Nikolaj Bjørner. Z3, an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. (Cited on pages 5 and 7.)

[44] Leonardo de Moura and Nikolaj Bjørner. Engineering dpll(t) + saturation. In *PROC. 4TH IJCAR*, 2008. (Cited on page 6.)

[45] Leonardo de Moura and Bruno Dutertre. Yices: An SMT Solver. `http://yices.csl.sri.com/`. (Cited on pages 8 and 33.)

[46] Leonardo Mendonça de Moura and Nikolaj Bjørner. Model-based theory combination. *Electronic Notes in Theoretical Computer Science*, 198(2):37–49, 2008. (Cited on page 5.)

[47] Leonardo Mendonça de Moura and Nikolaj Bjørner. Generalized, efficient array decision procedures. In *FMCAD*, pages 45–52. IEEE, 2009. (Cited on page 27.)

[48] Leonardo Mendonça de Moura and Dejan Jovanovic. A model-constructing satisfiability calculus. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *VMCAI*, volume 7737 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2013. (Cited on page 66.)

[49] Nachum Dershowitz. Orderings for term rewriting systems. *Theoretical Computer Science*, 17(3):279–301, March 1982. (Cited on page 79.)

[50] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–320. North-Holland, 1990. (Cited on page 78.)

[51] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52:365–473, May 2005. (Cited on page 7.)

[52] David L. Dill. A retrospective on mur*phi*. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 77–88. Springer, 2008. (Cited on page 1.)

[53] I. Dillig, T. Dillig, and A. Aiken. Cuts from proofs: A complete and practical technique for solving linear inequalities over integers. In *CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *LNCS*, pages 233–247. Springer, 2009. (Cited on pages 72 and 74.)

[54] Claire Dross, Sylvain Conchon, Johannes Kanig, and Andrei Paskevich. Reasoning with triggers. In Fontaine and Goel [62]. (Cited on page 138.)

[55] B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In *CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *LNCS*, pages 81–94. Springer, 2006. (Cited on pages 46 and 74.)

[56] Bruno Dutertre and Leonardo de Moura. The Yices SMT solver. available at `http://yices.csl.sri.com/tool-paper.pdf`, 2006. (Cited on page 7.)

[57] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003. (Cited on page 9.)

[58] Niklas Een and Niklas Sörensson. An extensible sat-solver [ver 1.2], 2003. (Cited on pages 1 and 7.)

[59] G. Farkas. Über die theorie der einfachen ungleichungen. *Journal für die Reine und Angewandte Mathematik*, 124:1–27, 1902. (Cited on page 35.)

[60] Jean-Christophe Filliâtre, Sam Owre, Harald Rueß, and Natarajan Shankar. ICS: Integrated Canonization and Solving (Tool presentation). In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of CAV'2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 246–249. Springer, 2001. (Cited on pages 6 and 7.)

[61] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013. (Cited on page 10.)

[62] Pascal Fontaine and Amit Goel, editors. Manchester, UK, 2012. LORIA. (Cited on pages 155 and 156.)

[63] The Frama-C platform for static analysis of C programs, 2008. `http://www.frama-c.cea.fr/`. (Cited on pages 1 and 10.)

[64] Amit Goel, Sava Krstić, and Alexander Fuchs. Deciding array formulas with frugal axiom instantiation. In *Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning*, SMT '08/BPR '08, pages 12–17, New York, NY, USA, 2008. ACM. (Cited on pages 11, 27 and 138.)

[65] A. Griggio. A practical approach to satisability modulo linear integer arithmetic. *Journal on Satisfiability, Boolean Modeling and Computation*, 8:1–27, 2012. (Cited on page 74.)

[66] A. Griggio, B. Schaafsma, A. Cimatti, and R. Sebastiani. MathSAT 5: An SMT Solver for Formal Verification. `http://mathsat.fbk.eu//`. (Cited on pages 5, 8, 25, 33 and 72.)

[67] Jérôme Guitton, Johannes Kanig, and Yannick Moy. Why Hi-Lite Ada? In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 27–39, Wrocław, Poland, August 2011. (Cited on page 10.)

[68] Aarti Gupta, Malay K. Ganai, and Chao Wang. Sat-based verification methods and applications in hardware verification. In Marco Bernardo and Alessandro Cimatti, editors, *SFM*, volume 3965 of *Lecture Notes in Computer Science*, pages 108–143. Springer, 2006. (Cited on page 1.)

[69] Leopold Haller, Alberto Griggio, Martin Brain, and Daniel Kroening. Deciding floating-point logic with systematic abstraction. In Gianpiero Cabodi and Satnam Singh, editors, *FMCAD*, pages 131–140. IEEE, 2012. (Cited on page 149.)

[70] J.-M. Hullot. Associative commutative pattern matching. In *Proc. 6th IJCAI (Vol. I), Tokyo*, pages 406–412, August 1979. (Cited on page 30.)

[71] The ISABELLE system. `http://isabelle.in.tum.de/`. (Cited on page 1.)

[72] Jean-Pierre Jouannaud and Hélène Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal on Computing*, 15(4), November 1986. (Cited on page 122.)

[73] D. Jovanovic and L. de Moura. Cutting to the chase solving linear integer arithmetic. In *CADE-23, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, volume 6803 of *LNCS*, pages 338–353. Springer, 2011. (Cited on pages 72, 74 and 75.)

[74] Deepak Kapur. Shostak's congruence closure as completion. In H. Comon, editor, *Proceedings of the 8th International Conference on Rewriting Techniques and Applications*, volume 1232. Springer-Verlag, 1997. (Cited on pages 23 and 122.)

[75] Deepak Kapur and Calogero G. Zarba. A reduction approach to decision procedures, 2006. (Cited on page 27.)

[76] Leonid Khachiyan. Fourier-motzkin elimination method. In Christodoulos A. Floudas and Panos M. Pardalos, editors, *Encyclopedia of Optimization*, pages 1074–1077. Springer, 2009. (Cited on page 4.)

[77] Hyondeuk Kim, Fabio Somenzi, and HoonSang Jin. Efficient term-ite conversion for satisfiability modulo theories. In Oliver Kullmann, editor, *SAT*, volume 5584 of *Lecture Notes in Computer Science*, pages 195–208. Springer, 2009. (Cited on page 64.)

[78] Donald E. Knuth and Peter B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970. (Cited on page 79.)

[79] K. Korovin and A. Voronkov. Solving systems of linear inequalities by bound propagation. In *CADE-23, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, volume 6803 of *LNCS*, pages 369–383. Springer, 2011. (Cited on page 75.)

[80] D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 1 edition, 2008. (Cited on pages 25, 26 and 71.)

[81] Sava Krstić and Sylvain Conchon. Canonization for disjoint unions of theories. *Information and Computation*, 199(1-2):87–106, May 2005. (Cited on page 83.)

[82] Sava Krstic and Amit Goel. Architecting solvers for sat modulo theories: Nelsonoppen with dpll. frontiers of combining systems, 2007. (Cited on page 66.)

[83] Dallas S. Lankford. Canonical inference. Memo ATP-32, University of Texas at Austin, March 1975. (Cited on page 79.)

[84] Dallas S. Lankford and A. M. Ballantyne. Decision procedures for simple equational theories with permutative axioms: Complete sets of permutative reductions. Research Report Memo ATP-37, Department of Mathematics and Computer Science, University of Texas, Austin, Texas, USA, August 1977. (Cited on page 122.)

[85] Stéphane Lescuyer. *Formalisation et développement d'une tactique réflexive pour la démonstration automatique en Coq*. Thèse de doctorat, Université Paris-Sud, January 2011. (Cited on page 133.)

[86] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, July 1993. (Cited on page 1.)

[87] J. L. LIONS. Ariane 5 flight 501 failure. `http://www.ima.umn.edu/~arnold/disasters/ariane5rep.html`. (Cited on page 1.)

[88] David C. Luckham, Steven M. German, Friedrich W. von Henke, Richard A. Karp, P. W. Milne, Derek C. Oppen, Wolfgang Polak, and William L. Scherlis. Stanford pascal verifier user manual. Technical report, Stanford, CA, USA, 1979. (Cited on page 7.)

[89] Claude Marché. On ground AC-completion. In Ronald. V. Book, editor, *4th International Conference on Rewriting Techniques and Applications*, volume 488 of *Lecture Notes in Computer Science*, Como, Italy, April 1991. Springer. (Cited on pages 9 and 122.)

[90] Claude Marché. Normalized rewriting: an alternative to rewriting modulo a set of equations. *Journal of Symbolic Computation*, 21(3):253–288, 1996. (Cited on pages 77, 84, 118 and 122.)

[91] William McCune. Otter 3.3 reference manual. *CoRR*, cs.SC/0310056, 2003. (Cited on page 2.)

[92] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *ANNUAL ACM IEEE DESIGN AUTOMATION CONFERENCE*, pages 530–535. ACM, 2001. (Cited on pages 1, 3 and 7.)

[93] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. on Programming, Languages and Systems*, 1(2):245–257, October 1979. (Cited on page 7.)

[94] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27:356–364, 1980. (Cited on page 4.)

[95] Greg Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, 1981. `http://www.cs.washington.edu/education/courses/cse599f/06sp/papers/NelsonThesis.pdf`. (Cited on pages 6 and 27.)

[96] Greg Nelson. Techniques for program verification. Research Report CSL-81-10, Xerox Palo Alto Research Center, 1981. http://research.compaq.com/SRC/esc/Simplify.html. (Cited on page 7.)

[97] Robert Nieuwenhuis and Albert Oliveras. Fast Congruence Closure and Extensions. *Inf. Comput.*, 2005(4):557–580, 2007. (Cited on pages 4 and 23.)

[98] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Abstract dpll and abstract dpll modulo theories. In *In LPAR'04, LNAI 3452*, pages 36–50. Springer, 2005. (Cited on pages 4 and 66.)

[99] Robert Nieuwenhuis and Albert Rubio. A precedence-based total AC-compatible ordering. In Claude Kirchner, editor, *Proc. 5th Rewriting Techniques and Applications, Montréal, LNCS 690*. Springer, June 1993. (Cited on pages 82 and 85.)

[100] John O'Leary. Theorem proving in intel hardware design. In Ewen Denney, Dimitra Giannakopoulou, and Corina S. Pasareanu, editors, *NASA Formal Methods*, volume NASA/CP-2009-215407 of *NASA Conference Proceedings*, page 5, 2009. (Cited on page 1.)

[101] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752, Saratoga Springs, NY, June 1992. Springer. (Cited on pages 6 and 7.)

[102] Gerald E. Peterson and Mark E. Stickel. Complete sets of reductions for some equational theories. *Journal of the ACM*, 28(2):233–264, April 1981. (Cited on page 122.)

[103] Vaughan R. Pratt. Anatomy of the pentium bug. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *TAPSOFT*, volume 915 of *Lecture Notes in Computer Science*, pages 97–107. Springer, 1995. (Cited on page 1.)

[104] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, pages 4–13, New York, NY, USA, 1991. ACM. (Cited on pages 4, 8, 25, 26 and 53.)

[105] Silvio Ranise and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `http://smtcomp.sourceforge.net/`, 2006. (Cited on pages 7 and 22.)

[106] Alexandre Riazanov and Andrei Voronkov. The design and implementation of vampire. *AI Commun.*, 15(2-3):91–110, 2002. (Cited on page 2.)

[107] Robert E. Shostak. An algorithm for reasoning about equality. *Communications of the ACM*, 21(2):583–585, july 1978. (Cited on page 23.)

[108] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965. (Cited on page 3.)

[109] Harald Rueß and Natarajan Shankar. Deconstructing shostak. In *LICS* [1], pages 19–28. (Cited on page 6.)

[110] A. Schrijver. *Theory of linear and integer programming*. Wiley-Interscience series in discrete mathematics and optimization. John Wiley & sons, 1998. (Cited on pages 4, 8, 26, 35 and 75.)

[111] R. E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31:1–12, 1984. (Cited on pages 4, 6, 7 and 21.)

[112] Joso L Marques Silva. Grasp - a new search algorithm for satisfiability. pages 220–227, 1996. (Cited on pages 1, 3 and 7.)

[113] H. J. S. Smith. On systems of linear indeterminate equations and congruences. *Proceedings of the Royal Society of London*, 11:86–89, 1860. (Cited on page 36.)

[114] W. Snyder. Efficient completion: a $o(n.log(n))$ algorithm for generating reduced sets of ground rewrite rules equivalent to a set of ground equations e. In N. Dershowitz, editor, *Proc. 3rd Conf. on Rewriting Techniques and Applications*. Springer-Verlag, 1989. Lecture Notes in Computer Science. (Cited on page 4.)

[115] Jean Souyris and Denis Favre-Félix. Proof of properties in avionics. In Renè Jacquart, editor, *Building the Information Society*, volume 156 of *IFIP International Federation for Information Processing*, pages 527–535. Springer US, 2004. (Cited on pages 1 and 10.)

[116] Aaron Stump, Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for an extensional theory of arrays. In *LICS* [1], pages 29–37. (Cited on page 27.)

[117] G. Sutcliffe and C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998. (Cited on page 2.)

[118] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.3*, 2010. `http://coq.inria.fr`. (Cited on pages 1 and 127.)

[119] Nikolai Tillmann and Jonathan De Halleux. Parameterized unit testing with microsoft pex (long tutorial), 2010. (Cited on page 1.)

[120] Ashish Tiwari. Combining equational reasoning. In Silvio Ghilardi and Roberto Sebastiani, editors, *FroCos*, volume 5749 of *Lecture Notes in Computer Science*, pages 68–83, Trento, Italy, September 2009. Springer. (Cited on pages 10 and 122.)

[121] Emanuele Viola. E-unifiability via narrowing. In *Proceedings of the 7th Italian Conference on Theoretical Computer Science*, ICTCS '01, pages 426–438, London, UK, UK, 2001. Springer-Verlag. (Cited on page 114.)

[122] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. Spass version 3.5. In *Proceedings of the 22nd International Conference on Automated Deduction*, CADE-22, pages 140–145, Berlin, Heidelberg, 2009. Springer-Verlag. (Cited on page 2.)